SMPTE ST 2042-1:2009

SMPTE STANDARD

VC-2 Video Compression

Table of Contents

Page

Foreword 7
Intellectual Property 7
Introduction 7

1 Scope 8
2 Conformance Notation 8
3 Normative References 8
4 Definition of Acronyms and Terms 9
4.1 Acronyms 9
4.2 Terms 9
5 VC-2 Conventions 11
5.1 Naming Conventions: 11
5.2 State Representation 11
5.3 Number Formats 11
5.4 Data Types 11
5.4.1 Elementary Data Types 11
5.4.2 Compound Data Types 12
5.5 Functions and Operators 12
5.5.1 Assignment 12
5.5.2 Boolean Functions and Operators 13
5.5.3 Integer Functions and Operators 13
5.5.4 Array and Map Functions and Operators 14
5.5.5 Precedence and Associativity of Operators 15
5.6 Pseudocode 16
5.6.1 Processes and Functions 16
5.6.2 Variables 17
5.6.3 Control Flow 18
6 Overall VC-2 Specification 19
7 Video Formats 21
7.1 Color Model 21
7.2 Interlace 21
7.3 Component Sampling 21
7.4 Bit Resolution and Signal Ranges 22
7.5 Video Frame Size and Rate 22

Copyright © 2009 by THE SOCIETY OF
MOTION PICTURE AND TELEVISION ENGINEERS
3 Barker Avenue., White Plains, NY 10601
(914) 761-1100

Approved
November 2, 2009

8 Encoding Overview (Informative) 22
8.1 Picture Input Processing 23
8.2 Wavelet Transform 23
8.3 Division Into Subbands 23
8.4 DC Subband Prediction 24
8.5 Coefficient Scanning 25
8.5.1 Core Syntax Coefficient Scanning 25
8.5.2 Low Delay Coefficient Ordering and Scanning 25
8.6 Quantization and Quantizer Estimation 27
8.6.1 Quantization of the DC band 27
8.7 Data Encoding 27
8.7.1 Arithmetic coding 27
8.8 Stream Syntax 28
9 Decoding Overview 28
9.1 Decoding Functions 28
9.1.1 Functional Description 29
9.1.2 Data Decoding 29
9.1.2.1 VC-2 Data Codings 30
9.1.3 VC-2 Syntax Decoding 30
9.1.4 Subband Decoding 30
9.1.4.1 DC Band Prediction 31
9.1.5 Inverse Quantization 31
9.1.5.1 Quantizer Factor and Offset (Informative) 31
9.1.6 Coefficient Coding Order 32
9.1.7 Inverse Discrete Wavelet Transform 32
9.1.7.1 Wavelet Filter Support 33
9.1.8 Clipping 33
10 VC-2 Stream 34
10.1 Pseudocode 34
10.2 VC-2 Stream Syntax 34
10.3 VC-2 Sequence Syntax 34
10.3.1 Parse Info Headers 35
10.3.2 Data Units 36
10.3.2.1 Auxiliary Data 36
10.3.2.2 Padding 36
10.4 Parse Info Header Syntax 37
10.4.1 Parse Codes 38
10.4.1.1 Parse Code Values (Informative) 39
10.5 VC-2 Sequence Decoding (Informative) 39
10.5.1 Non Sequential Picture Decoding (Informative) 40
11 Sequence Header 41
11.1 Parse Parameters 43
11.1.1 Version Number 43
11.1.2 Profiles and Levels 43
11.2 Base Video Format 44
11.3 Source Parameters 45
11.3.1 Setting Source Defaults 46

11.3.2 Frame Size 46
11.3.3 Color difference Sampling Format 47
11.3.4 Scan Format 47
11.3.5 Frame Rate 48
11.3.6 Pixel Aspect Ratio 50
11.3.7 Clean Area 51
11.3.8 Signal Range 52
11.3.9 Color Specification 53
11.3.9.1 Color Primaries 54
11.3.9.2 Color Matrix 55
11.3.9.3 Transfer Function 55
11.4 Picture Coding Mode 56
11.5 Initializing Coding Parameters 56
11.5.1 Picture Dimensions 57
11.5.2 Video Depth 57
12 Picture Syntax 58
12.1 Picture Header 58
12.2 Wavelet Transform 58
12.3 Transform Parameters 59
12.3.1 Wavelet Filter 60
12.3.2 Transform Depth 60
12.3.3 Codeblock Parameters (Core Syntax Only) 60
12.3.4 Slice Coding Parameters (Low Delay Syntax Only) 62
12.3.4.1 Slice Parameters 62
12.3.4.2 Quantization Matrices 62
13 Transform Data Syntax 63
13.1 Subband Data Structure 63
13.1.1 Wavelet Data Initialization 64
13.1.2 Wavelet Subband Dimensions 65
13.2 Inverse Quantization 65
13.2.1 Quantization Factors and Offsets 66
13.3 DC Subband Prediction 67
13.4 Core Syntax Wavelet Coefficient Unpacking 68
13.4.1 Core Syntax Transform Data 68
13.4.2 Subbands 69
13.4.2.1 Zero Subband 70
13.4.2.2 Non-Skipped Subbands 70
13.4.3 Subband Codeblocks 70
13.4.3.1 Codeblock Dimensions 71
13.4.3.2 Codeblock Unpacking Loop 71
13.4.3.3 Skipped Codeblock Flag 71
13.4.3.4 Codeblock Quantizer Offset 72
13.4.4 Subband Coefficients 72
13.4.4.1 Zero Parent 73
13.4.4.2 Zero Neighborhood 74
13.4.4.3 Sign Prediction 74
13.4.4.4 Coefficient Context Selection 75

13.5 VC-2 Low Delay Wavelet Coefficient Unpacking 77
13.5.1 Overall Process 77
13.5.2 Slice Unpacking 78
13.5.3 Determining the Number of Bytes in a Slice 79
13.5.4 Setting Slice Quantizers 79
13.5.5 Slice Subbands 80
13.5.5.1 Slice Subband Area 80
13.5.5.2 Luma Slice Subband Data 80
13.5.5.3 Color difference Slice Subband Data 81
14 Picture Decoding 81
14.1 Overall Picture Decoding Process 81
14.2 Picture IDWT 82
14.3 Component IDWT 82
14.3.1 Vertical and Horizontal Synthesis 83
14.3.2 One-Dimensional Synthesis 84
14.3.2.1 Mathematical Formulation of Lifting Processes (Informative) 86
14.3.3 Lifting Filter Parameters 87
14.3.4 Removal of IDWT Pad Values 90
14.4 Picture Output Ranges 90
Annex A VC-2 Data Coding Definitions (Normative) 92
A.1 Bit Packing and Data Input 92
A.1.1 Reading a Byte 92
A.1.2 Reading a Bit 92
A.1.3 Byte Alignment 92
A.2 Fixed Length Data 93
A.2.1 Boolean 93
A.2.2 n-bit Unsigned Integer Literal 93
A.2.3 n-byte Unsigned Integer Literal 93
A.3 Variable-length Codes 94
A.3.1 Data Input for Bounded Block Operation 94
A.3.2 Unsigned Interleaved Exp-Golomb Codes 95
A.3.3 Signed Interleaved Exp-Golomb Codes 97
A.4 Parsing of Arithmetic Coded Data 98
A.4.1 Context Probabilities 98
A.4.2 Arithmetic Decoding of Boolean Values 99
A.4.3 Arithmetic Decoding of Integer Values 99
A.4.3.1 Binary Coding and Contexts 99
A.4.3.2 Unsigned Integer Decoding 99
A.4.3.3 Signed Integer Decoding 100
Annex B Arithmetic Coding (Normative) 101
B.1 Arithmetic Coding Principles (Informative) 101
B.1.1 Interval Division and Scaling 101
B.1.2 Finite Precision Arithmetic 102
B.1.3 Symbol Probability Estimation 102
B.2 Arithmetic Decoding Engine (Normative) 104
B.2.1 State and Contexts 104
B.2.2 Initialization 104

B.2.3 Data Input 105
B.2.4 Decoding Boolean Values 105
B.2.5 Renormalization 105
B.2.6 Updating Context Probabilities 106
B.3 Arithmetic Encoding (Informative) 108
B.3.1 Encoder Variables 108
B.3.2 Initialization 108
B.3.3 Encoding Binary Values 108
B.3.3.1 Scaling the Interval 108
B.3.3.2 Updating Contexts 109
B.3.3.3 Renormalization and Output 109
B.3.4 Flushing the Encoder 109
Annex C Predefined Video Formats (Normative) 111
Annex D Profiles and Levels (Normative) 115
D.1 Profiles 115
D.1.1 Low Delay Profile 115
D.1.2 Simple Profile 115
D.1.3 Main Profile 116
D.2 Levels 116
Annex E Low Delay Quantization Matrices (Normative) 117
E.1 Default Quantization Matrices 117
E.2 Quantization Matrix Design and Quantizer Selection (Informative) 118
E.2.1 Noise Power Normalization 118
E.2.2 Custom Quantization Matrices 120
Annex F Video Systems Model (Informative) 122
F.1 Color Models 122
F.1.1 YC_BC_R Coding 122
F.1.2 YC_OC_G Coding 122
F.1.3 Signal Range 122
F.1.4 Color Primaries 123
F.1.5 Color Matrix 123
F.2 Transfer Characteristics 123
F.2.1 TV Transfer Characteristic 123
F.2.2 Extended Color Gamut 124
F.2.3 Linear 124
F.3 Frame Rate 124
F.4 Aspect Ratios And Clean Area 124
F.4.1 Pixel Aspect Ratio 124
F.4.1.1 Using Non-Square Pixel Aspect Ratios 124
F.4.2 Clean Area 125
Annex G Wavelet Decimation and Reconstruction Processes (Informative) 126
G.1 Overview of Wavelet Processing 126
G.2 The Lifting Process 128
Annex H Bibliography (Informative) 130

Figures

Figure 8.1 – Sample encoder functional block diagram 22
Figure 8.2 – A single DWT transform stage 23
Figure 8.3 – Identification of the subbands of a 4-level wavelet transform 24
Figure 8.4 – Division of subbands into codeblocks, illustrating the constant codeblock structure at each level....25
Figure 8.5a – Formation of a single 4x4 slice from 2-level transform coefficients 26
Figure 8.5b – An 8x4 array of slices for a 2-level wavelet transform....26
Figure 8.6 – Arithmetic encoding process 28
Figure 9.1 – Functional VC-2 decoder block diagram 29
Figure 9.2 – Prediction aperture for DC bands....31
Figure 9.3 – Wavelet Decoding Steps 33
Figure 10.1 – VC-2 stream 34
Figure 10.2 – Overview of VC-2 sequence structure....35
Figure 10.3 – Parse Info header syntax 37
Figure 11.1 – Sequence Header....42
Figure 11.2 – Parse Parameters....43
Figure 11.3 – Source Parameters....46
Figure 11.4 – Frame Size 46
Figure 11.5 – Sampling Format 47
Figure 11.6 – Scan format 48
Figure 11.7 – Frame rate 49
Figure 11.8 – Pixel Aspect Ratio 50
Figure 11.9 – Clean area 51
Figure 11.10 – Signal range 52
Figure 11.11 – Color specification syntax....54
Figure 12.1 – VC-2 picture....58
Figure 12.2 – Wavelet transform data 59
Figure 12.3 – Transform parameters 59
Figure 12.4 – Spatial partitioning (core syntax only) 61
Figure 12.5 – Slice parameters....62
Figure 13.1 – 4-Level subband array....64
Figure E.1 – Subband weights for a 1-level decomposition 119
Figure E.2 – Subband weights for a 2-level decomposition 120
Figure G.1 – Single wavelet processing stage comprising decimation and reconstruction filters....126
Figure G.2 – Illustration of the alias frequency generation and cancellation in a wavelet filter bank....127
Figure G.3 – Two-step wavelet processing filter bank....127
Figure G.4 – Decomposition of a single image into 7 wavelet frequency bands....128
Figure G.5 – Decomposition of the EBU “Boats” picture into 7 wavelet frequency bands....128
Figure G.6 – Polyphase representation of wavelet filter banks 129

Foreword

SMPTE (the Society of Motion Picture and Television Engineers) is an internationally recognized standards-developing organization. Headquartered and incorporated in the United States of America, SMPTE has members in over 80 countries on six continents. SMPTE's Engineering Documents, including Standards, Recommended Practices and Engineering Guidelines, are prepared by SMPTE's Technology Committees. Participation in these Committees is open to all with a bona fide interest in their work. SMPTE cooperates closely with other standards-developing organizations, including ISO, IEC and ITU.

SMPTE Engineering Documents are drafted in accordance with the rules given in Part XIII of its Administrative Practices. SMPTE ST 2042-1 was prepared by Technology Committee 10E.

Intellectual Property

At the time of publication no notice had been received by SMPTE claiming patent rights essential to the implementation of this Standard. However, attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. SMPTE shall not be held responsible for identifying any or all such patent rights.

Introduction

This section is entirely informative and does not form an integral part of this document.

The VC-2 standard specifies the compressed stream syntax and reference decoder operations for a video compression system. VC-2 is an intra frame video compression system aimed at professional applications that provides efficient coding at many resolutions including various flavors of CIF, SDTV and HDTV. VC-2 utilises wavelet transforms that decompose the video signal into frequency bands. The codec is designed to be simple and flexible, yet be able to operate across a wide range of resolutions and application domains.

The system provides the following capabilities:

- **Multi-resolution transforms**. Data is encoded using the wavelet transform, and packed into the bitstream subband by subband. High compression ratios result in a gradual loss of resolution. Lower resolution output pictures can be obtained by extracting only the lower resolution data.
- **Dual syntax**. A low delay syntax is available for applications requiring very low, fixed, latency. This can be of the order of a few lines of input or output video. The low delay syntax is suitable for light compression for the re-use of low bandwidth infrastructure, for example carrying HDTV over SD-SDI links. The low delay syntax uses simple Variable Length Codes for entropy codes. The core syntax provides much greater compression efficiency at the expense of a whole picture delay. The core syntax can use a highly efficient form of binary adaptive arithmetic coding for maximum performance.
- **Frame and field coding**. Both frames and fields can be individually coded.
- **CBR and VBR operation**. VC-2 permits both constant bit rate and variable bit rate operation. When the low delay syntax is used, the bit rate will be constant for each area (VC-2 slice) in a picture to ensure constant latency.
- **Variable bit depths**. 8, 10, 12 and 16 bit formats and beyond are supported.
- **Multiple color difference sampling formats**. 444, 422 and 420 video are all supported.
- **Lossless and RGB coding**. A common toolset is used for both lossy and lossless coding. RGB coding is supported via the YCoCg integer color transform.
- **Wavelet filters**. A range of wavelet filters can be used to trade off performance against complexity. The Daubechies (9,7) filter is supported for compatibility with JPEG2000. A Fidelity filter is provided for improved resolution scalability.
- **Simple stream navigation**. The encoded stream forms a doubly-linked list with each picture header indicating an offset to the previous and next picture, to support field-accurate high-speed navigation with no parsing or decoding required.

1 Scope

This standard defines the VC-2 video compression system through the stream syntax, entropy coding, coefficient unpacking process and picture decoding process. The decoder operations are defined by means of a mixture of pseudo-code and mathematical operations.

VC-2 is an intra frame video codec that uses wavelet transforms together with entropy coding that can be readily implemented in hardware or software at very high bit rates. Additional standards and recommended practices may define specific constraints on the encoding for particular applications.

2 Conformance Notation

Normative text is text that describes elements of the design that are indispensable or contains the conformance language keywords: "shall", "should", or "may". Informative text is text that is potentially helpful to the user, but not indispensable, and can be removed, changed, or added editorially without affecting interoperability. Informative text does not contain any conformance keywords.

All text in this document is, by default, normative, except: the Introduction, any section explicitly labeled as "Informative" or individual paragraphs that start with "Note:"

The keywords "shall" and "shall not" indicate requirements strictly to be followed in order to conform to the document and from which no deviation is permitted

The keywords, "should" and "should not" indicate that, among several possibilities, one is recommended as particularly suitable, without mentioning or excluding others; or that a certain course of action is preferred but not necessarily required; or that (in the negative form) a certain possibility or course of action is deprecated but not prohibited.

The keywords "may" and "need not" indicate courses of action permissible within the limits of the document.

The keyword “reserved” indicates a provision that is not defined at this time, shall not be used, and may be defined in the future. The keyword “forbidden” indicates “reserved” and in addition indicates that the provision will never be defined in the future.

A conformant implementation according to this document is one that includes all mandatory provisions ("shall") and, if implemented, all recommended provisions ("should") as described. A conformant implementation need not implement optional provisions ("may") and need not implement them as described.

Unless otherwise specified the order of precedence of the types of normative information in this document shall be as follows. Normative prose shall be the authoritative definition. Tables shall be next, followed by formal languages, then figures, and then any other language forms.

3 Normative References

The following standards contain provisions which, through reference in this text, constitute provisions of this standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this standard are encouraged to investigate the possibility of applying the most recent edition of the standards indicated below.

ITU-R BT.601-6 (01/07), Studio Encoding Parameters of Digital Television for Standard 4:3 and Wide-Screen 16:9 Aspect Ratios

ITU-R BT.709-5 (04/02), Parameter Values for the HDTV Standards for Production and International Programme Exchange

ITU-R BT.1361 (02/98), Worldwide Unified Colorimetry and Related Characteristics of Future Television and Imaging Systems

ITU-T Recommendation H.264 (03/09) | ISO/IEC 14496-10:2008, Advanced Video Coding (AVC)

SMPTE 428-1-2006, D-Cinema Distribution Master (DCDM) — Image Characteristics

SMPTE 2036-1-2007, Ultra High Definition Television — Image Parameter Values for Program Production.

4 Definition of Acronyms and Terms

This section defines the acronyms and terms used in the VC-2 specification.

4.1 Acronyms

4CIF: Four times CIF
4SIF: Four times SIF
CIE: Commission internationale de l'éclairage (International Commission on Illumination)
CIF: Common Image Format a.k.a. SIF 625
CSF: Contrast Sensitivity Function
DC: Direct Current
DWT: Discrete Wavelet Transform
FIR: Finite Impulse Response
HD(TV): High Definition (Television)
HH: High-High (of a subband)
HL: High-Low (of a subband)
IDWT: Inverse Discrete Wavelet Transform
ITU: International Telecommunications Union
LH: Low-High (of a subband)
LL: Low-Low (of a subband)
LS(B): Least Significant (Bit)
LUT: Look Up Table
MS(B): Most Significant (Bit)
NTSC: National Television Systems Committee
QCIF: Quarter Common Image Format
QSIF: Quarter Source Image Format
SD(TV): Standard Definition (Television)
SIF: Source Input Format
VC: Video Codec
VLC: Variable Length Code

4.2 Terms

AC (sub)band: Any signal band that is not the DC subband.

Arithmetic coding: A form of entropy coding used by VC-2, which is used in addition to exp-Golomb coding.

Color difference: The term color difference is the direct equivalent to ‘luma’ (see ‘luma’ definition below). In this standard, the term color difference is used to cover both gamma-corrected and non-gamma-corrected signals. In some cases, the term ‘difference’ has been truncated to ‘diff’ to save space (e.g. color_difference to ‘color_diff’).

Codeblock: A rectangular array of wavelet coefficients within a component subband.

Codec: A truncation of the terms “coder” and “decoder”.

DC prediction: The prediction of coefficients within the dc subband from neighbouring coefficients.

DC (sub)band: The signal band that represents data composed from the lowest frequency band of a wavelet transform (0-LL).

Discrete Wavelet Transform (DWT): A means of transforming an array of values into frequency components through the iterated application of a filter bank.

Entropy coding: A term for describing any mathematical process that is intended to reduce the number of bits needed to encode data in a lossless manner.

Exp-Golomb: A form of variable length entropy coding. The VC-2 specification uses an interleaved variant of exp-Golomb coding, see Annex A.3 for more information.

Intra Frame Coding: The coding of frames of video without reference to any other frames.

Inverse Discrete Wavelet Transform (IDWT): The inverse of the DWT that converts an array of frequency components back into an array of values.

Inverse quantization: Mapping a quantized value to a representative value for the sub-range indicated by the quantized value.

Lifting: A technique for implementing a DWT in a computationally efficient and reversible manner. (Note: See Annex A Bibliography item “Ripples in Mathematics”, chapter 3, for more information.)

Low delay: A term used to define the VC-2 mode that can be used to compress video with a delay of less than one frame duration.

Luma: The weighted sum of RGB components of color video, which may or may not be gamma-corrected. (This term is used to prevent confusion with the term ‘luminance’ that is created only from linear light levels as used in color science.)

Parse info header: A component of the VC-2 stream that provides information on how to parse the stream. It supports navigating through the stream without having to decode every frame.

Parsing: A process by which text strings and numerical values within data are recognized and used to provide syntactic meaning.

Picture: A single frame or field of video.

Quantization: The division of the range of coefficient values into a number of sub-ranges, each of which is represented by quantized value.

Raster scan: Any two-dimensional array of samples, that are scanned left to right, then top to bottom.

Sequence: The data contained in a VC-2 sequence corresponds to a single video sequence with constant video parameters as defined in Section 9. A VC-2 sequence is preceded by a parse info header that indicates the beginning of the sequence with a unique parse code. A VC-2 sequence can be extracted from a VC-2 stream and decoded as in independent entity.

Slice: A component part of the low delay syntax that provides for compression of small parts (slices) of a picture in order to reduce delay.

State: The set of current decoder variable values.

Stream: A concatenation of VC-2 sequences.

Subband: The signal band that represents data composed from a single frequency band of a wavelet transform.

5 VC-2 Conventions

This section defines the VC-2 conventions including the naming conventions, the decoder state representation, number formats, data types, arithmetic operations, variables, logical operations, order of operator precedence and pseudocode conventions.

5.1 Naming Conventions:

Syntax names are expressed as single text strings in a monotype font (such as Courier) with any word spacing using the underscore character. For all syntax names:

`plain_text` is used for variables,

`bold text` is used for state and control flow and,

`italicized_text` is used for functions.

5.2 State Representation

This standard uses a state model to express parsing and decoding operations. The state of the decoder/parser shall be stored in the variable **`state`**. Individual elements of the decoder state (state variables) may be accessed by means of named labels, e.g. **`state`**`[var_name]` (i.e. **`state`** is a map, as defined in Section 5.4.2).

The decoder state shall be globally accessible within the decoder. Other variables, not declared as inputs to a process, shall be local to that process.

The parsing and decoding operations are defined in terms of modifying the decoder state. The state variables need not directly correspond to elements of the stream, but may be calculated from them taking into account the decoder state as a whole. For example, a state variable value may be differentially encoded with respect to another value, with the difference, not the variable itself, encoded in the stream.

The VC-2 stream syntax structure is illustrated with informative parse diagrams that complement the normative stream syntax definitions. The parsing process is defined by means of pseudocode and/or mathematical formulae. The conventions for these elements are described in the following sections.

5.3 Number Formats

Numbers without a prefix shall be interpreted as decimal numbers.

Numbers with the prefix ‘b’ indicates that the following value shall be interpreted as a binary natural number (non-negative integer).

Example: The value b1110100 is equal to the decimal value 116.

Numbers with the prefix ‘0x’ indicates the following value shall be interpreted as a hexadecimal (base 16) natural number.

Example The value 0x7A is equal to the decimal value 122.

5.4 Data Types

5.4.1 Elementary Data Types

Three basic types shall be used in the pseudocode:

- **Boolean** – A Boolean variable shall have only two possible values: **True** and **False**.
- **Integer** – A positive or negative whole number or zero.
- **Label** – A unique immutable value used in control structures and to access maps (see below). Labels are not given values in this standard. The value assigned to each label is implementation dependent but shall be unique within the implementation.

Variants of the basic types shall be as defined below:

- **UInt** – An unsigned integer that is a non-negative (>=0) number of arbitrary magnitude.
- **UIntnn** – An unsigned integer that requires nn bits to express the possible range of values. For example UInt8 indicates a 8 bit integer constrained to lie in the range 0 to 255.
- **Int** – A signed integer that is a number of arbitrary magnitude.
- **Intnn** – A signed integer that requires nn bits to express the possible range of values. For example UInt32 indicates a 32 bit signed integer constrained to lie in the range -2147483648 to +2147483647.

At various places in this specification informative notes may be included to indicate the type or range of variables.

5.4.2 Compound Data Types

Elementary and compound data types may be aggregated into a single compound data type.

There shall be three compound data types:

- **Set** - A collection of data types. A set is indicated by enclosing the elements within curly braces, for example `{a,b,c}` represents a set containing the values `a,b` and `c`. An empty set is indicated by `{}`. The usual set-theoretic operations such as: $\cup$ (union), $\cap$ (intersection), $\in$ (membership) apply to sets and the other compound data types.
- **Map** – A collection (set) of data types whose elements are accessed by their corresponding label. For example, `p[Y] ,p[C1] ,p[C2]` might be the values of the different video components of a pixel. The set of labels corresponding to the elements of a map `m` can be accessed by `args(m)`, so that, for example, `args(p)` returns `{Y,C1,C2}`.
- **Array** – A collection of data types accessed by a non-negative integer index. This compound data type is typically used to represent an array of variables. Elements of a 1-dimensional array `a` are accessed by `a[n]` for `n` in the range 0 to `length(a) - 1`.

A compound data type may contain other compound data types. For example, a two dimensional array is an array of one dimensional arrays. Elements of a 2-dimensional array are accessed by `a[n][m]` for `0 <=` m `<= (width(a)-1)`, and `0 <=` n `<= (height(a)-1)`. Compound data types may also be more complex. For example, picture data, `pic`, may be considered to be a map of arrays, where `pic[Y]` is a 2-dimensional array storing luma data, and `pic[C1]` and `pic[C2]` are two-dimensional arrays storing color difference data.

Elements may be added to a map or array by assignment using the appropriate index (label or integer). For example, `a[7]=2`, adds element 7 to the array `a`, if `a` does not already contain element 7, then this element is assigned the value 2.

5.5 Functions and Operators

This section defines the functions and operators used in the pseudocode in this specification. Functions and operators are similar, though functions use the syntax, function(argument 1, argument2,…), whereas operators are simply placed before or between operands, e.g. a + b. The difference is purely syntactic and is to correspond with conventional mathematical notation.

5.5.1 Assignment

The assignment operation (=) applies to all variable types. After performing the assignment operation:

```
a=b
```

the value of `a` shall become equal to that of `b`, and the value of `b` shall remain unchanged. For a set (or map or array) this constitutes an element-wise copy; i.e.,

```
a[x]=b[x]
```

for all valid values of x.

5.5.2 Boolean Functions and Operators

The following functions and operators are defined for one or more Boolean arguments:

not (`not a`) shall return **True** for a boolean value `a` if and only if `a` is **False**

and `(a and b)` shall return **True** if and only if `a` and `b` are both **True**. Operator "and" may be used in pseudocode conditions to denote the logical AND between Boolean values, for example:

```
if (condition1 and condition2):
    …etc.
```

or `(a or b)` shall return **True** if either `a` or `b` are **True**, else it returns **False.** Operator "or" may be used in pseudocode conditions to denote the logical OR between Boolean values, for example:

```
if (condition1 or condition2):
 … etc.
```

5.5.3 Integer Functions and Operators

The following functions and operators are defined for one or more numerical arguments.

Absolute value	`|a| = a` if `(a >= 0)` else `|a| = -a`
Sign	`sign(a) = 1` if `a>0, 0` if `a=0`, and `-1` if `a<0`
Addition	The sum of `a` and `b` shall be represented by `a+b`
Subtraction	`a` minus `b` shall be represented by `a-b`.
Multiplication	`a` times `b` shall be represented by `a*b`.
Integer division	Integer division is defined for integer values `a` and `b`, `b>0` where: `n=a//b` shall be defined to be the largest number such that `n * b ≤ a` i.e. numbers are rounded towards -infinity. Note: this differs from C/C++ conventions of round towards 0.
Remainder	For integers `a, b`, with `b > 0`, the remainder `a%b` shall be defined by: `a%b = a-(a//b)*b` where `a%b` shall always satisfy the condition `0 ≤ (a%b) < b`
Exponentiation	For integer values `a, b`>0, exponentiation shall be denoted by a^b.and is equal to `a*a*…*a*a` (`b` times). a^0 is defined to be 1 for all `a`.
Maximum	`max(a, b)` shall return the largest value of `a` and `b`
Minimum	`min(a, b)` shall return the smallest of values `a` and `b`
Clip	`clip(a,b,t)` shall clip the value a to the range defined by `b` and `t` such that: `clip(a,b,t) = min(max(a,b),t)`
Shift down (>>)	For integers `a`, `b`, with `b>=0`, `a>>b` shall be defined as $a//2^b$
Shift up (<<)	For integers `a`, `b`, with `b>=0`, `a<<b` shall be defined as $a*2^b$.
Integer logarithm	$m=intlog_2(n)$, for `n>1`, `m` shall be the integer such that $2^{m-1}<n\leq 2^m$ e.g. $intlog_2(25)=intlog_2(32)=5$.

Mean Given a set, $S = \{s_0, s_1, \ldots, s_{n-1}\}$ of integer values, the integer unbiased mean, `mean(S)`, shall be defined as:

$$mean(S) = (s_0 + s_1 + \ldots + s_{n-1} + (n//2))//n$$

The following bitwise operations are defined for non-negative integer values:

& Logical *and* shall be applied between the corresponding bits in the 2's complement binary representation of two numbers, e.g. 13&6 is b1101 & b110, which equals b100, or 4.

| Logical *or* shall be applied between the corresponding bits in the 2's complement binary representation of two numbers, e.g. 13|6 is b1101 | b110, which equals b1111, or 15.

^ Logical *xor* shall be applied between the corresponding bits in the 2's complement binary representation of two numbers, e.g. 13 ^ 6 is b1101 ^ b110, which equals b1011, or 11.

&= `a &= b` is equivalent to `a = (a & b)`.

^= `a ^= b` is equivalent to `a = (a ^ b)`.

|= `a |= b` is equivalent to `a = (a | b)`.

Note: Bitwise-not is not defined for integers to avoid ambiguity concerning leading zeroes.

The following logical operators are defined for integer arguments:

== Test of equality of two variables. `a == b` is **True** if and only if the value of `a` equals the value of `b`

< Less than

<= Less than or equal to

> Greater than

>= Greater than or equal to

!= Not equal to. a != b is equivalent to not (a == b)

The following combined assignment operators are defined for integer arguments:

Operators +, -, *, //, %, >>, <<, &, |, ^, may be combined with the assignment operator (as for the Boolean operators &, |, and ^ above). For example:

+= `a += b` is equivalent to `a = (a + b)`

-= `a -= b` is equivalent to `a = (a - b)`

5.5.4 Array and Map Functions and Operators

The following functions and operators are defined for arrays and maps.

Indexing For an array or map `a`, `a[index]` shall return an element of `a`. If `a` is a map the index shall be a label, else if `a` is an array the index shall be an integer.

Scalar Assignment Where the notation `a=0` is used for an array of integer values, it

shall mean: “set all elements of the array to zero”.

Insertion `a[index] = b` shall insert a copy of `b` into set a if the element does not already exist.

Arguments for a map `a`, `arg(a)` shall return the set of the indexing labels.

Length for a one dimensional array `a`, `length(a)` shall return the number of elements in the array.

Width for a two dimensional array `a`, `width(a)` shall return the width the array. The width shall be the number of scalar elements corresponding to the right most array index.

Height for a two dimensional array `a`, `height(a)` shall return the height the array. The height shall be the number of one dimensional arrays in the two dimensional array and the height dimension shall correspond to the left most array index.

5.5.5 Precedence and Associativity of Operators

To avoid any confusion over the order of operator precedence, every equation makes extensive use of the expression operators “(“ and “)”. All operations shall recursively execute the innermost expression(s) first until the calculation has been completed. In cases where the expression operators do not make clear the order of precedence, Table 5.1 shall define the descending order of operator precedence and the associativity of each operator (i.e., the topmost operator has greatest precedence).

Table 5.1 – Operator precedence and associativity

Operator Precedence	Associativity
( ) [ ]	left to right
* //	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
! (not)	right to left
& (and)	left to right
^ (xor)	left to right
|	left to right
= += -= *= //= &= ^= |= <<= >>=	right to left

5.6 Pseudocode

Most of the normative specification is defined by means of pseudocode. The syntax is intended to be both precise and descriptive. The pseudocode is not intended to form the basis for the implementation of a VC-2 decoder.

All processing defined by this standard is precise and the entire specification can be implemented using only the data types, functions and operators defined herein. That is, no operations on "real" or "floating point" numbers are required. All operations shall be implemented with sufficiently large integers so that overflow cannot occur.

The type of variables in the pseudocode is not explicitly declared. A variable assumes a type when it is assigned a value, which shall always have a defined type.

Pseudocode in this standard is sometimes accompanied by a figure illustrating the content of the code.

5.6.1 Processes and Functions

Decoding and parsing operations are specified by means of processes - a series of operations acting on input data and the state variables. A process can also be a function, which means it returns a value, but it need not do so.

So a process taking arguments `in1` and `in2` looks like:

```
foo(in1, in2):
    op1(in1)
    op2(in2)
    ...
```

While a function process looks like:

```
bar(in1, in2):
   op1(in1)
   foo(in1, in2)
   ...
   return out1
```

5.6.2 Variables

All input variables shall be deemed to be passed *by reference* in this specification. This means that any modification to a variable value that occurs within a process shall also apply to that variable within the calling process *even if it has a different name* in the calling process. This differs from the conventions of common programming languages such as C, C++ and Java, and care must be taken in reading this standard.

Note: One way to understand this is to envisage variable names as labels for pointers to workspace memory.

For example, if we define *foo* and *bar* by:

```
foo():
   num = 0
   bar(num)
   state[var_name] = num
```

and:

```
bar(val):
   val = val + 1
```

then at the end of *foo*, **state**[var_name] has been set to 1.

If a process is particularly complex, it may be broken into a number of steps with intermediate discussion. This is signaled by appending and pre-pending "…" to the parts of the pseudocode specification:

```
foo():
   code
   ...
```

[text]

```
...
   more code
...
```

[text]

```
...
   even more code
```

The intervening text may define or modify variables used in the succeeding pseudocode, and shall be considered as a normative part of the specification of the process. This is done as it is sometimes much clearer to split up a long and complicated process into a number of steps.

5.6.3 Control Flow

The pseudocode shall comprise a series of statements, linked by functions and flow control statements such as `if`, `while`, and `for`.

The statements do not have a termination character, unlike the ";" in C for example. Blocks of statements shall be indicated by indentation: where indenting in shall begin a block and indenting out shall end the block.

Statements that expect a block (and hence a following indentation) shall end in a colon.

`if:`

The **`if`** control shall evaluate a boolean or a boolean function, and if **True**, shall pass the flow to the following statement or block of statements. If the control evaluates as **False**, then there is an option to include one or more **`else if`** controls which offer alternative responses if some other condition is **True**. If none of the preceding controls evaluate to **True**, then there is the option to include an **`else`** control that catches remaining cases.

```
...
   if (control1):
      block1
   else if (control2):
      block2
   else if (control3):
      block3
   else:
      block4
```

The **`if`** and **`else if`** conditions shall be evaluated in the order in which they are presented.

For example, if `control1` or `control2` is **True** in the preceding example, `block3` will not be executed even if `control3` is **True**; neither will *`block`*`4`.

`for:`

The **`for`** control shall repeat a loop over an integer range of values. For example,

```
...
   for i = 0 to (n - 1):
      foo(i)
```

calls *`foo`*`()` with value `i`, as `i` steps through from `0` to `n - 1` inclusive.

`for each:`

The **for each** control shall loop over the elements in a list in order. For example,

```
...
   for each c in Y, C1, C2:
      foo(c)
```

calls *foo*(Y), then *foo*(C1), then *foo*(C2).

while:

The **while** control shall repeat a loop so long as a switch variable is true. When it is false, the shall loop break to the next statement(s) outside the block.

```
...
   while (condition):
      block1
   block2
```

6 Overall VC-2 Specification

The VC-2 specification includes provisions for both the Core Syntax and Low Delay variants.

- The Core syntax codes entire pictures (fields or frames), uses exp-Golomb and/or arithmetic coding (as a post process) with quantization at picture level and uses codeblocks for efficient coding. The core syntax achieves the best compression but results in at least one picture of delay. It is also suitable for lossless coding.
- The Low Delay syntax codes picture slices for low delay, uses exp-Golomb coding only and defines quantization at slice level. It can achieve very low delay (or latency) but at the expense of less efficient compression. It is not suitable for lossless coding.

The VC-2 specification comprises the following parts:

Section #	Title	Description
7	Video Formats	A specification of the video formats supported by VC-2.
8	Encoding Overview (Informative)	An informative overview of the processes used by a VC-2 encoder to compress the video. The sequence of functional elements in the encoder, and the methods used for realizing them, are implementation-dependent. The purpose of the encoder is to compress a video input to produce a bitstream that complies with the normative VC-2 stream syntax.
9	Decoding Overview	An overview of VC-2 decoder operations, providing the starting point for subsequent sections.
10	VC-2 Stream	A specification for the top-level parsing and decoding of data units in a VC-2 stream.
11	Sequence Header	A definition of the metadata contained in the sequence header, necessary for a compliant decoder to parse the remainder of the stream and access picture data.
12	Picture Syntax	A definition of the picture data unit syntax and semantics, including all metadata necessary for configuring wavelet transforms and coefficient decoding in both core and low delay modes.
13	Transform Data Syntax	A definition of the processes needed to unpack and dequantize the wavelet coefficient data from the VC-2 stream in both core and low delay modes, ready for handing over to the picture decoder.
14	Picture Decoding	The specification of the picture decoder that takes decoded wavelet transform data, decoding it to produce viewable images.

The VC-2 specification is supported by a number of normative annexes that provide core information required for the system definition.

Annex A	VC-2 Data Coding Definitions	Defines the VC-2 data coding definitions for wavelet coefficients and header elements.
Annex B	Arithmetic Coding	Provides a description of arithmetic coding, a definition of the arithmetic decoding engine and functions used in the VC-2 core syntax and a description of a VC-2 compatible arithmetic encoder.
Annex C	Predefined Video Formats	Defines the parameters for a range of base video formats. The listing in this annex does not prevent other video format specifications from being used, but provides a short-cut method for identifying commonly used video formats.
Annex D	Profiles and Levels	Defines the VC-2 Profiles and describes the use of VC-2 Levels and how they relate to Profiles.
Annex E	Low Delay Quantization Matrices	Defines the Quantization Matrices for the low delay syntax.

The remaining annexes are informative and provide additional information that can aid implementers but do not provide any further normative provisions of this standard.

7 Video Formats

This section defines the video formats supported by this standard.

A selection of widely used video formats are defined in normative Annex C. These video formats are characterized by their widespread use in television, cinema and multimedia applications.

This list is not exhaustive, however, and VC-2 is a general purpose video compression system. These predefined formats are base formats that may be modified element by element to support a much larger range of possible video formats. Support is provided by the sequence parameters of the bitstream (Section 10) for signaling both the base video format and any modifications for complete characterization of the video format metadata.

7.1 Color Model

VC-2 supports any video format that codes the raw image colors in a luma (grey-level) component with two associated color difference components. These components are referred to as Y, C_1 and C_2.

In ITU defined systems (including ITU-R BT.709 and ITU-R BT.1361), the Y, C_1 and C_2 values shall relate to the E'_Y, E'_U and E'_V video components respectively. These video components are also widely referred to as Y, U, V and Y, C_B, C_R.

In the ITU-T H.264 reversible color transform, the Y, C_1 and C_2 values shall correspond to the video components Y, C_O, C_G.

Note: Coding using Y, C_O, C_G. provides a simple reversible conversion to and from R-G-B components by using lossless integer transforms. The use of Y, C_O, C_G supports lossless coding of RGB video and allows VC-2 to be treated as an RGB compression system for applications that require this feature.

7.2 Interlace

VC-2 supports both interlace and progressive video formats.

VC-2 codes pictures where a picture may be a frame or a field. A frame contains lines of spatial information of a video signal.

For progressive video, lines contain samples starting from one time instant and continuing through successive lines to the bottom of the frame.

For interlaced video, a frame consists of two fields, a top field and a bottom field. Each field is the assembly of alternate lines of a frame. One of these fields will commence one field period later than the other (i.e., the fields may be coded as either top field first or bottom field first). Each line of a top field is spatially located immediately above the corresponding line of the bottom field. Conversely, each line of a bottom field is spatially located immediately below the corresponding line of the top field.

A pair of fields constituting a frame may correspond to distinct time intervals (true interlace scanning) or to the same time interval (progressive segmented frames).

7.3 Component Sampling

Color difference components C_1 and C_2 may be coded with the same dimensions as the Y component (4:4:4) sampling, or with half-width (4:2:2) or half-dimension (4:2:0) sampling.

Y, C_1 and C_2 picture components shall be sampled at the same temporal instant.

Note: All pictures are considered as individual entities whether or not all lines were sampled at the same instant. In video sequences that are not frame-based, such as 30fps interlaced video carrying 24fps progressive images in a 3:2 pull-down sequence, the compression performance might not be optimum. In such cases, a pre-processor could provide an encoder with a more easily compressed source such as the original 24fps source pictures. Such pre-processing does not form any part of this standard.

7.4 Bit Resolution and Signal Ranges

The bit depth of each component sample is, in principle, unrestricted. Application-specific codecs may restrict the supported bit depth to a single value or a limited range of values.

Video is represented internally within the decoder specification as a bipolar (signed integer) signal. Video is presented at the video interface as an unsigned integer value by addition of an offset to these values (Section 13.4). Metadata concerning black level and white level is transmitted within the data stream (Section 11.3.8), but is not enforced at the decoder video interface: output video may undershoot or overshoot these values.

7.5 Video Frame Size and Rate

The frame size and frame rate are, in principle, unrestricted. Application-specific codecs may restrict both the supported frame size and frame rate to a single value or a limited range of values. See Annex D.2.

8 Encoding Overview (Informative)

This section identifies the modules of the VC-2 picture encoder and describes the function of each module. It provides guidance for implementers through a generalized description of the encoder. It does not normatively define the construction of a VC-2 encoder. The encoder must create a VC-2 stream that complies with the provisions of the normative decoder specification (Sections 9-14). This section is wholly informative and contains no normative provisions.

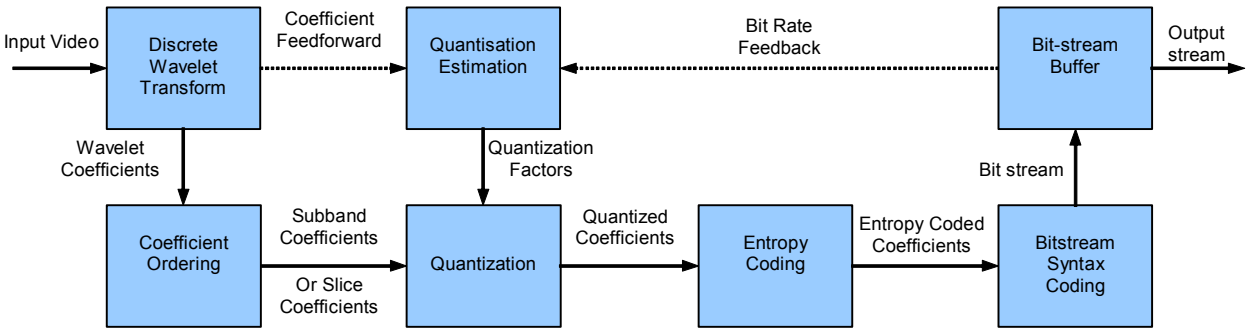

Figure 8.1 – Sample encoder functional block diagram

The video picture data is transformed in both the horizontal and vertical axes by a Discrete Wavelet Transform into an array of transform coefficients. These coefficients are then re-ordered into subbands or slices and quantized.

An implementation-dependent quantization estimation process determines the quantizers based on application requirements. No quantizer estimation processes are defined by this standard. Figure 8.1 illustrates two possible quantizer estimation sources (“coefficient feedforward” and “bit rate feedback”).

The quantized samples are entropy encoded to reduce the output bit rate. Entropy coding always implements exp-Golomb coding and, optionally, arithmetic coding as a post-process.

The entropy coded coefficients are encapsulated and formatted with VC-2 signaling information to create a stream that complies with the VC-2 syntax defined in Sections 10-13.

Note: For some applications, and for some methods of deriving the quantization factors, a buffer can be used to achieve a constant bit rate stream. The buffer, the rate control scheme and the method for deriving the quantization factors are beyond the scope of this standard.

8.1 Picture Input Processing

The encoder operates independently on input video pictures, which can be frames or fields depending on the configuration of the encoder, and so constitutes a separate processing path.

Depending on the picture source format, the color difference components can be sub-sampled relative to the luma component.

8.2 Wavelet Transform

The encoder uses a Discrete Wavelet Transform (DWT) to separate the frequency bands of each video component.

The wavelet transform can be thought of as being applied to a picture component as a whole, conceived of as a two-dimensional array of values. For the core syntax, it is generally necessary to calculate the wavelet transform of the whole picture before quantization can occur. So, for the core syntax, the encoding delay (latency) would be greater than one active picture (i.e., excluding any blanking in the input signal).

In order to calculate the wavelet coefficients corresponding to an area of the picture only a few lines from the vicinity of that area need be known. The low delay syntax takes advantage of this to support low delay encoding. In the low delay syntax, the coefficients for slices at the top of the picture can be processed and output before input pixels for the bottom of the picture are available to the encoder. In this way encoding delays as low as a few lines can be achieved.

A single DWT stage, in one dimension, is a filter bank that decomposes a component signal into pair of signals that represent the lower and upper frequencies of the source signal. The transform process can be applied in successive stages, where each stage is applied to the low frequency band of the previous transform, resulting in a series of transforms that represent the signal components in octave frequency bands. This same process can be applied in the two dimensions of an image source by applying both a vertical and horizontal filtering step in each stage, resulting in data that is decomposed in both the horizontal and vertical dimensions to produce four spatial frequency bands. The “low-low” frequency band of the previous transform stage can be further decomposed, analogously with the one dimensional case. The process of the DWT and the resulting transformed image is illustrated in Annex G.

8.3 Division Into Subbands

The DWT creates a series of subbands for each picture, representing spatial frequency bands.

In a single transform stage, illustrated in Figure 8.2, a picture component is first filtered horizontally by high-pass and low-pass filters to produce low and high frequency subbands. These subbands are subsampled by a factor of two so that there is no increase in the number of samples. Then both these subbands are themselves filtered vertically and subsampled in the same way, to produce four subbands labeled LL, HL, LH and HH and carrying respectively: low-horizontal, low-vertical; high-horizontal, low-vertical; low-horizontal, high-vertical and high horizontal, high vertical frequencies.

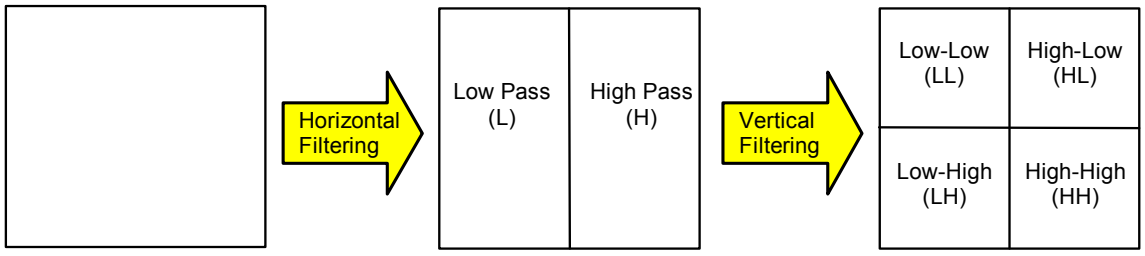

Figure 8.2 – A single DWT transform stage

In a multi-level transform, the LL band at each stage is filtered and subsampled in the same way, resulting in a hierarchy of subbands. In the case of a 4-level transform, the results are as illustrated in Figure 8.3.

The resulting subbands are identified by level and orientation, as shown in Figure 8.3. The 0-LL subband is the lowest frequency subband, the only subband that carries a DC component, and thus appears as a small, but viewable, picture. The HL, LH and HH bands occur at all other wavelet levels. As a result of the iterative filtering process, successive levels carry increasingly higher frequencies and bandwidth.

In principle, there is no limit to the depth of a wavelet transform, subject to the coding level constraints of Annex D, and to the requirement that there is an integral number of coefficients in the 0-LL subband.

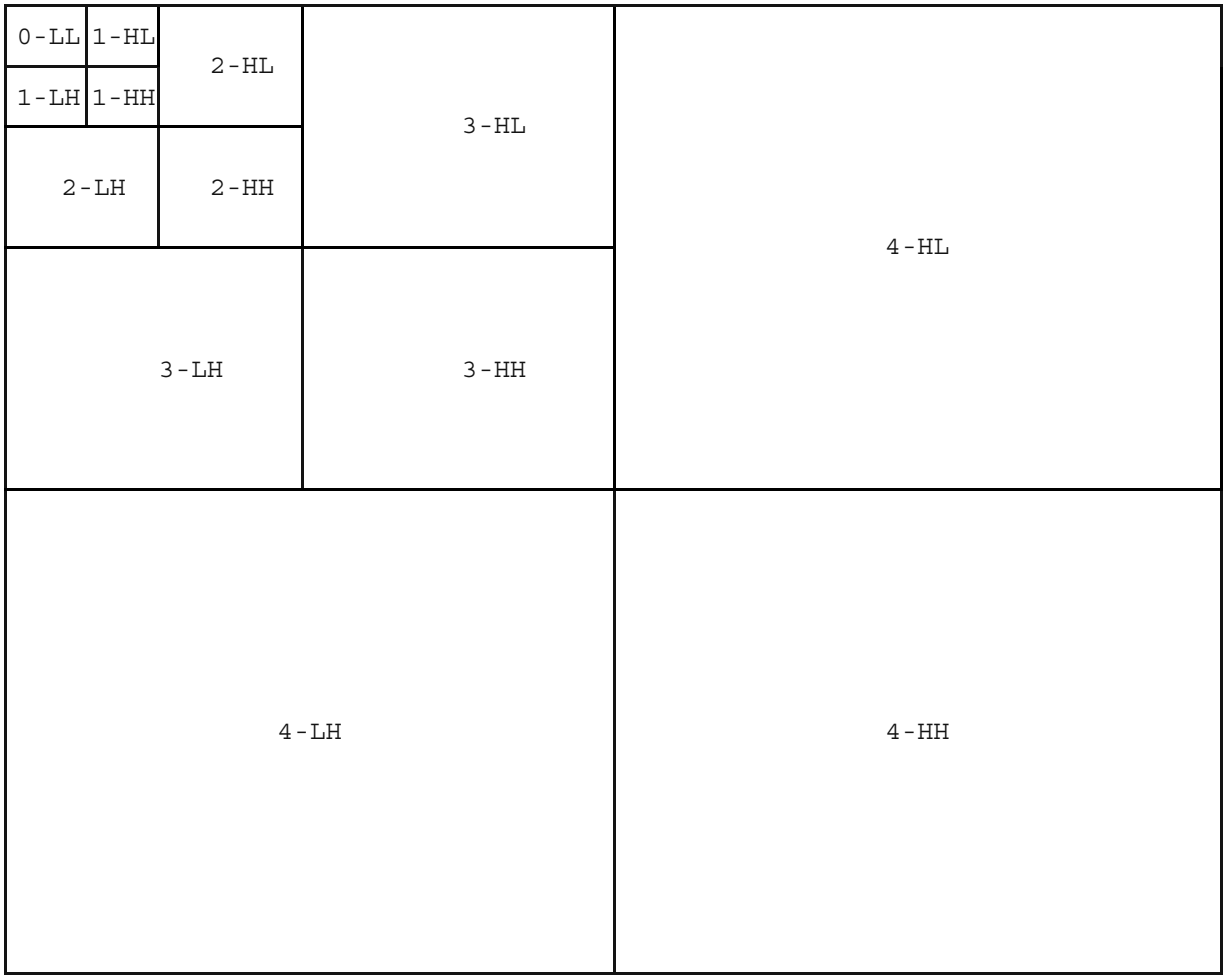

Figure 8.3 – Identification of the subbands of a 4-level wavelet transform

8.4 DC Subband Prediction

Spatial prediction is applied to the 0-LL (DC) subband only. For each pixel in the 0-LL subband, it provides a 3-pixel prediction to improve the coding efficiency. The prediction process is defined in Section 13.3.

8.5 Coefficient Scanning

8.5.1 Core Syntax Coefficient Scanning

In the core syntax, subbands are determined by applying the wavelet transform to each picture component as a whole picture.

Core syntax subbands can optionally be divided into codeblocks,which are rectangular arrays of subband coefficients (Figure 8.4). Where codeblocks are present, each subband is then raster-scanned both at the codeblock level and within each codeblock. The number of codeblocks vertically and horizontally is defined independently for each level. Where codeblocks are not present, the whole subband is coded in raster-scanned order. The subbands are coded in level order, and in the order HL, LH, HH for levels >0.

By default, there is a single quantization factor per subband. The encoder can optionally provide individual quantization factors for each codeblock.

Note: The use of codeblocks that are less than the subband array size provides for more localized scanning, which can be used to isolate 'busy' areas of a picture and improve coding efficiency. They are particularly useful in the higher frequency subbands.

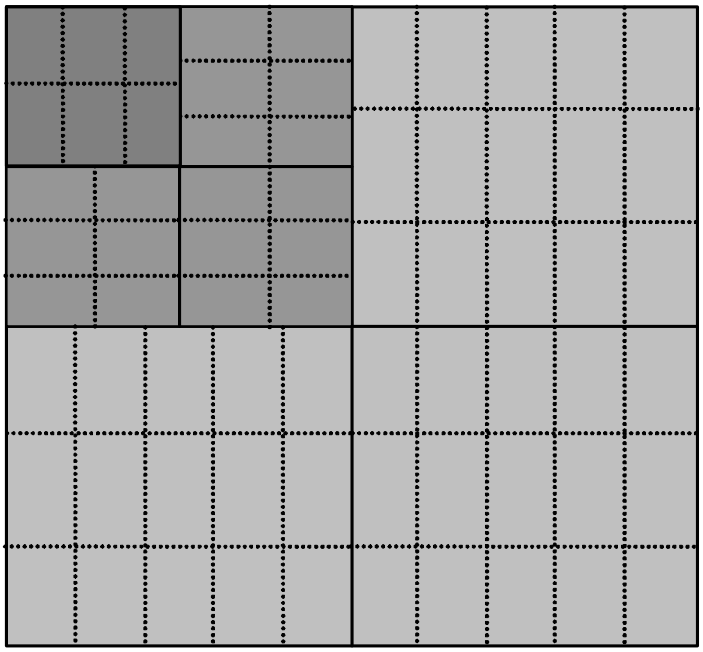

Figure 8.4 – Division of subbands into codeblocks, illustrating the constant codeblock structure at each level

8.5.2 Low Delay Coefficient Ordering and Scanning

In the low delay syntax, the coefficients for each subband are reordered into an array of 'local' wavelet transforms. Each local transform corresponds approximately to a frequency decomposition of a region of the picture. Each slice contains data from all three video components. This use of slices permits decoding of small areas of the picture thus reducing the coding and decoding delay. Figures 8.5a and 8.5b illustrate an example of forming and arranging slices, for a two-level transform.

Slice subbands correspond to codeblocks, and a slice partition corresponds to a codeblock partition in which the number of codeblocks is the same at all levels.

In the low delay syntax, each slice is scanned and coded in turn, in raster order, with each slice subband for each video component coded before the next slice.

In the low delay syntax it is the transform coefficients that are partitioned into slices, not the input pixels. This is in contrast to block transform compression systems, which partition the input pixels prior to performing the transform. As a result, slice coefficients can depend on picture samples adjacent to the nominal slice boundaries as well as within it. Thus slices correspond to a series of overlapping transforms on the picture data.

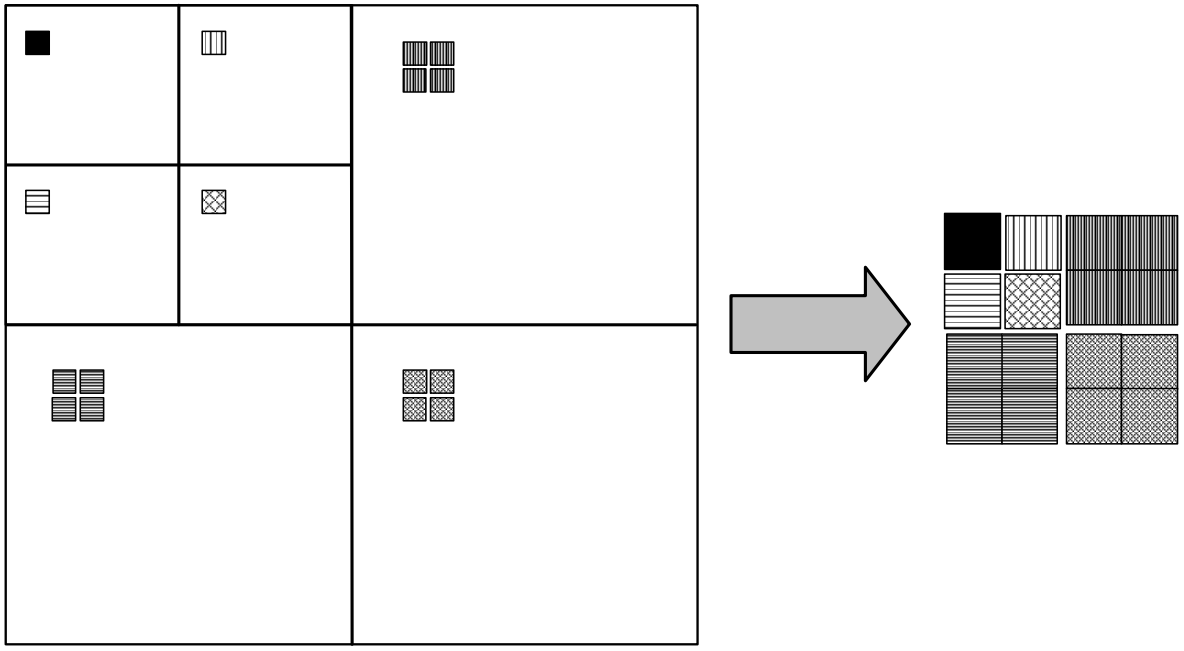

Figure 8.5a – Formation of a single 4x4 slice from 2-level transform coefficients

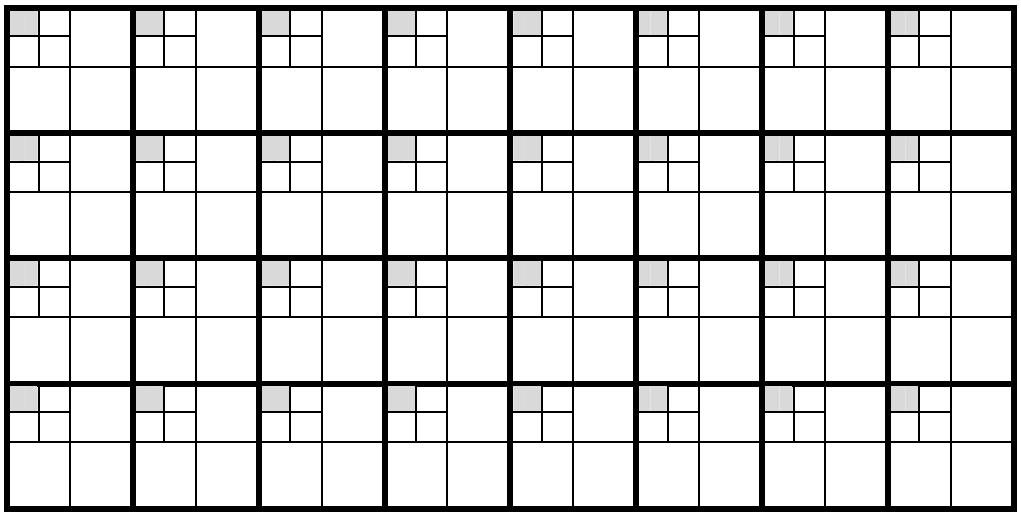

Figure 8.5b – An 8x4 array of slices for a 2-level wavelet transform

Note: In the case of the Haar wavelet transform, the order in which the transform and the slice partitioning are performed can be interchanged. That is, for the Haar wavelet transform only, VC-2 can be implemented as a block transform system.

8.6 Quantization and Quantizer Estimation

This specification only defines the inverse quantization process. Inverse quantization is defined in Section 13.2. Conceptually, forward quantization can be regarded as the integer division of the transform coefficients by a quantization factor.

Quantization factors, determined by the encoder, are encoded into the VC-2 stream to allow inverse quantization at the decoder. The process of determining quantization factors by the encoder is implementation specific and is beyond the scope of this specification.

The quantization factors are coded as quantization indices. A quantization index of zero indicates no quantization. Each increment of the quantization index indicates an increase in quantization by one quarter bit. For example, a quantization index of 4 means one bit of quantization or a (rounded) division by two.

For the core syntax, a VC-2 encoder provides a quantizer index for each subband. Where the subband is divided into more than one codeblock, a separate quantization index can be provided for each codeblock.

For the low delay syntax, a VC-2 encoder provides a quantizer index for each slice. Quantization indexes for each subband within a slice are derived by means of a quantization matrix.

8.6.1 Quantization of the DC band

In the DC subband, the encoded coefficients are prediction residues after spatial prediction, and quantization is applied to these prediction residues. To eliminate drift between encoder and decoder, an encoder must use fully reconstructed (inverse-quantized and inverse-predicted) coefficients for predicting any subsequent coefficients.

Care is required in setting the level of DC subband quantizers as spatial prediction can combine with temporal effects to produce visible artifacts.

8.7 Data Encoding

A variety of methods are employed for encoding data within the stream. These employ three basic data coding methods: fixed-length codes, variable-length codes and arithmetic coding.

The variable-length codes used are the interleaved form of exp-Golomb coding (Annex A.3).

Metadata needed to describe the video, and specify coding parameters is encoded using fixed-length and variable length codes, defined in annexes A.2 and A.3. Using variable length codes allows VC-2 to support a very wide range of video formats and coding parameters without an excessive bit rate overhead. Using VLCs will also allow the size of tables to be extended without a change of syntax, in any future revisions or extensions of this standard.

Wavelet coefficients are coded using variable-length codes and optionally using arithmetic coding. In the low delay syntax arithmetic coding is not used, and in the core syntax arithmetic coding can be used. The variable length codes used for coefficient coding are the same as those used for metadata, except that they are modified so as to be used in data blocks of a known size. This modification inserts bits of value 1 when data within a block has been exhausted. This has the effect of allowing a run of zero coefficients at the end of a data block to be decoded without signaling any further bits.

8.7.1 Arithmetic coding

The principles of arithmetic coding are described in Annex B.1. Annex B.3 describes an adaptive arithmetic coding engine that will produce a compliant bitstream when used in conjunction with the correct methods for binarization and context selection (described below).

Arithmetic coding is employed on integer-valued wavelet coefficients and uses a two-stage process as shown in Figure 8.6. First, integer values are binarized into a sequence of bits or boolean values. At the same time a “context” is generated for each of these bits. These boolean values, together with the corresponding context, are then coded by the binary arithmetic coding engine described in Annex B.3.

The binary arithmetic encoder requires a probability estimate for each bit it encodes. These are maintained for a set of contexts, or configurations of previously coded data values, and updated once a value has been coded.

The VC-2 binarization process also uses the interleaved exp-Golomb variable length code. Binary arithmetic coding can therefore be viewed as an optional post-process after variable-length coding.

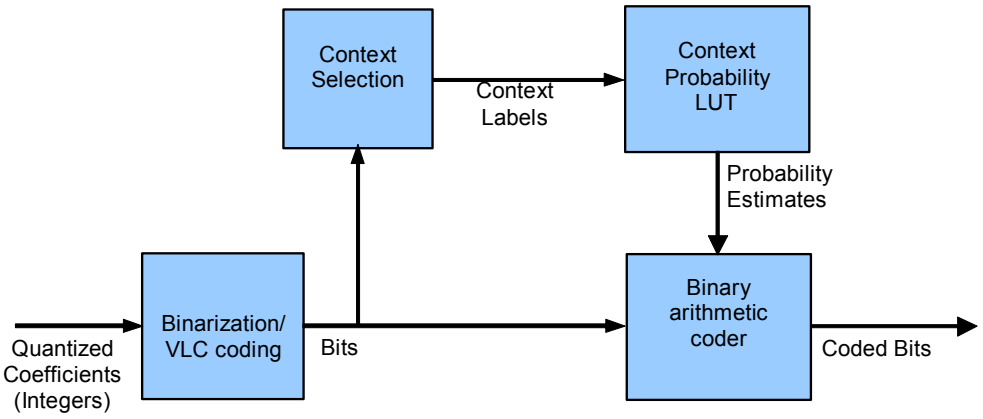

Figure 8.6 – Arithmetic encoding process

At low bit rates, arithmetic coding provides a substantial reduction in bit rate at the expense of additional decoder complexity. At high bit rates and for lossless coding, arithmetic coding yields a relatively small reduction in bit rate. For some applications it can be advantageous to forego this slightly lower bit rate to achieve lower complexity.

8.8 Stream Syntax

The VC-2 stream syntax (section 10) defines the structure of the VC-2 stream and the data types of all data within the stream.

9 Decoding Overview

This section presents an overview of the VC-2 decoding process that provides the starting point for the next five sections. It defines the modules of the decoder and describes the functional definitions of each module.

9.1 Decoding Functions

Figure 9.1 illustrates the main functional units in a VC-2 decoder.

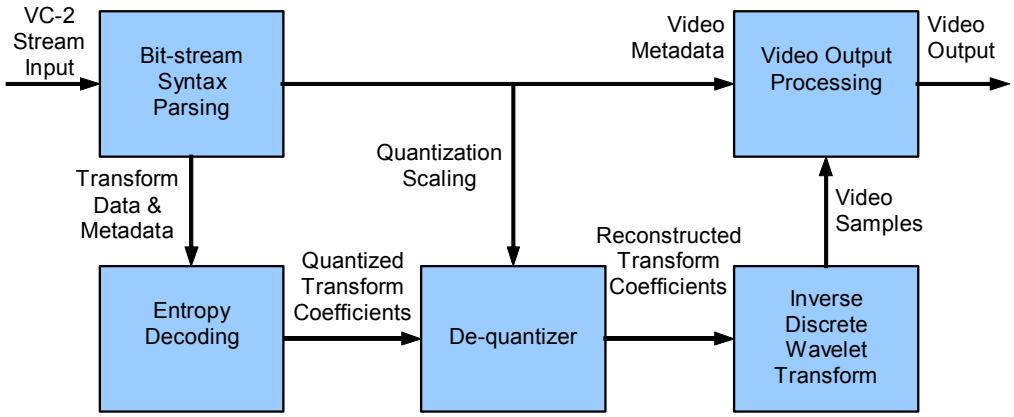

Figure 9.1 – Functional VC-2 decoder block diagram

The VC-2 stream syntax is defined in section 10 (and subsequent Sections 11-13), allowing a decoder to extract (unpack) the subband coefficients and the associated metadata necessary for successful decoding.

The data decoding processes shall be as defined by Section 9.1.2.

The inverse discrete wavelet transform decoding and video output processing processes shall be as defined in Section 13.

9.1.1 Functional Description

The following list defines the core functions of a VC-2 decoder.

- The VC-2 stream at the input to a VC-2 decoder shall comply with the stream syntax defined in Sections 10-13.
- The entropy encoded data within the VC-2 stream shall be decoded to create the coefficients for each subband together with all other data including quantization values and video metadata.
- The de-quantizer process shall be used to re-create transform coefficient samples. The transform samples shall then be input to the inverse DWT for conversion back to picture samples.

9.1.2 Data Decoding

The VC-2 stream shall comprise the sequence of data coded according the VC-2 stream syntax and the transform data as defined in Section 10. Different data coding methods are employed in different parts of the VC-2 stream.

Each element of the VC-2 stream shall be decoded according to one of the provisions defined in Annex A. These include:

- Bit-packing and byte alignment elements,
- A Boolean element (values of **True**[=1] and **False**[=0]),
- N-bit literal (a defined number of bits as an unsigned integer, msb first),
- N-byte literal (a defined number of bytes including single byte literals),
- Unsigned interleaved exp_Golomb codes,
- Signed interleaved exp_Golomb codes,
- Arithmetic coding

9.1.2.1 VC-2 Data Codings

Stream syntax header data shall use fixed-length and variable-length codes as defined in Annex A.2 and A.3. The variable-length code (VLC) used shall be an interleaved form of exp-Golomb coding.

In the core syntax, wavelet coefficient metadata shall be VLC-encoded. Wavelet coefficients shall be identified as using either VLCs or arithmetic coding as defined in Annex A.3. The binary arithmetic coding engine employed shall be as defined in Annex B.

In the low delay syntax, VLCs shall be used to code slice wavelet coefficients as well as wavelet metadata.

All syntax elements within the VC-2 stream shall be decoded as follows:

- Fixed length codes shall be decoded using:
  - *read_bool()*: read a single boolean value
  - *read_boolb()*: read a single boolean value from a bounded block
  - *read_nbits(n)*: read n bits
  - *read_uint_lit(n)*: read n bytes
- Variable length codes (for header data) shall be coded using:
  - *read_uint*(): read an unsigned integer
  - *read_sint*(): read a signed integer
- Variable length codes (for coefficient data) shall be decoded using:
  - *read_uintb*(): read an unsigned integer from a bounded block
  - *read_sintb*(): read a signed integer from a bounded block
- Arithmetic coded coefficient data shall be decoded by:
  - *read_boola*(): read an arithmetic coded boolean value
  - *read_uinta()*: read an arithmetic coded unsigned integer
  - *read_sinta()*: read arithmetic coded signed integer

9.1.3 VC-2 Syntax Decoding

The VC-2 syntax elements, defined in Section 11, constitute the metadata that describes the characteristics of the source pictures and the compression coding and data formatting parameters. The syntax elements that are needed to decode the compressed data shall be stored in a set of state variables within the decoder and these state variables shall be used by the decoder to extract and decode the transform data to form the output picture.

9.1.4 Subband Decoding

Figure 8.3 in Section 8 illustrates identifies the subbands and their frequency characteristics.

With the exception of the 0-LL band, the coefficient decoding process for each subband is dependent on data from the subbands of the next lower depth level. A *parent* subband shall be that subband of a given orientation in the next lower depth level and a *child* subband shall be that subband of the same orientation in the next higher depth level. When using arithmetic coding, decoding a coefficient in a child subband of a given orientation requires the prior decoding of the parent coefficient in the parent subband with the same orientation (where one exists).

For the core syntax, a VC-2 decoder may decode the subbands for a picture as a whole.

For the low delay syntax, a VC-2 decoder may decode each subband within a slice and then each slice within a picture as defined in Section 13.5.2.

Note: When using the low delay syntax, in the specific case of the Haar filter, the inverse transform can be performed on each slice in isolation. If other wavelet filters are used, then each decoded slice will need to access to

adjacent unpacked slice data to reconstruct picture data correctly.

9.1.4.1 DC Band Prediction

For both the core and low delay syntax, spatial prediction shall be applied to the 0-LL (DC) subband only. For each coefficient in the 0-LL subband, the prediction shall be defined as follows:

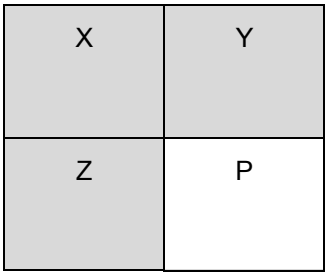

Figure 9.2 – Prediction aperture for DC bands

The prediction of the value of coefficient P shall be calculated as follows:

- Where coefficients X, Y and Z are available, coefficient P shall be predicted to be the mean value of surrounding pixels X, Y and Z.
- Where P is on the first line, it shall be predicted as the value of Z alone.
- Where P is the first sample in the row, it shall be predicted as the value of Y alone.
- Where P is the first sample in the row and on the first line only, the predicted value shall be zero.

Coefficient P shall be calculated as the sum of the prediction plus the value reconstructed from the stream.

The VC-2 syntax for DC subband prediction shall be as defined in Section 13.3.

9.1.5 Inverse Quantization

The inverse VC-2 quantization process shall be as defined in Section 13.2.

For the core syntax at least one quantizer index shall be present for each VC-2 subband. Where a subband is divided into more than one codeblock, each codeblock shall use a codeblock quantization index value where present, else it shall use the quantization index value defined for the subband.

For the low delay syntax a quantizer shall be provided for each slice, from which a quantizer for each slice subband shall be derived by means of a quantization matrix.

9.1.5.1 Quantizer Factor and Offset (Informative)

The quantizer factor can be considered to be the divisor in the quantization process in the encoder. The value of the quantizer factor is an integer approximation to:

```
4*(2**(qi/4))
```

where `qi` is the quantizer index. The pre-multiplication by 4 avoids large fractional changes in the quantization factor for small changes of the quantization index. The value of the quantizer factor is defined by integer operations only.

For a quantizer index of q, the quantized value of x would be 4x//quantizer_factor, which is approximately:

```
x/(2**(q/4))
```

The precise details of the quantization process performed by an encoder are outside the scope of this specification and are left to the encoder implementer.

In the decoder, the quantizer offset value is added to the quantized value before multiplication by the quantizer factor to provide a more accurate approximation to the original value.

The quantizer offset is usually the quantizer factor divided by two, except for low values of the quantizer index: the offsets having been selected so as to make inverse quantization and re-quantization by the same quantization factor transparent i.e. in this case the following two signal chains are equivalent:

1. Quantize -> Inverse Quantize -> (Re-)Quantize
2. Quantize

This property allows for coding with no multi-generational loss, provided that the same quantization index can be selected by each encoding stage. This can be done by minimizing quantization error in subsequent stages.

9.1.6 Coefficient Coding Order

Wavelet coefficients shall be ordered in subband order in the stream. The first subband shall be the DC subband (0-LL) followed by each successive level, from x = 1 to max, in the order x-HL, x-LH, x-HH, where ‘max’ is the level of wavelet decimation.

In the core syntax, the following rules shall apply:

- video components shall be coded sequentially in the order Y, C_1, C_2.
- each subband shall be coded as a separate block of data. Optionally the subbands may be partitioned into multiple "codeblocks". A subband that has not been partitioned may be regarded as a single codeblock. Coefficients within a subband shall be coded in raster scan order of codeblocks. That is, codeblocks are taken sequentially from left to right, starting on the first row of codeblocks then scanning subsequent rows to form a sequence of codeblocks. The coefficients within each codeblock shall be coded in raster scan order. That is, coefficients are taken sequentially from left to right, starting on the first line then scanning subsequent lines, to form a sequence of coefficients.
- at least one quantizer index shall be coded per subband. The encoder may optionally provide individual quantizer indices for each codeblock in the VC-2 stream.

In the low delay syntax, the following rules shall apply:

- wavelet coefficients from all subbands shall be spatially partitioned into slices. Within each slice, all luma coefficients shall be coded first, followed by all color difference coefficients, interleaving C_1 and C_2 component data coefficient by coefficient. Coding within a slice shall be in subband order and in raster scan order within a subband.
- a single quantization index shall be coded per slice.

9.1.7 Inverse Discrete Wavelet Transform

The decoder shall use the Inverse Discrete Wavelet Transform (IDWT) to re-combine the frequency bands of each video component.

Figure 9.3 illustrates the progressive nature of the IWDT process on a 2-stage wavelet coded picture.

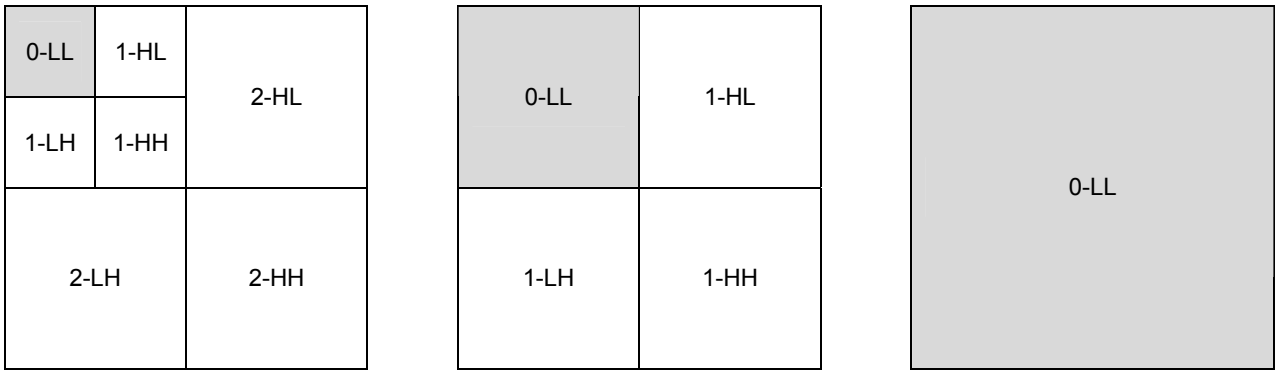

Figure 9.3 – Wavelet Decoding Steps

The leftmost picture represents a wavelet coded picture with all the wavelet coefficients grouped together (as wavelet bands 0-LL, 1-HL/LH/HH and 2-HL/LH/HH).

The first stage of reconstruction shall rebuild a new 0-LL band in two steps as follows:

1. Combine 0-LL and 1-LH to form an intermediate picture 0-L, and combine 1-HL and l-HH to form an intermediate picture 0-H
2. Combine intermediate pictures 0-L and 0-H to reconstruct a new 0-LL picture.

The result is a new 0-LL picture of twice the original 0-LL picture size as illustrated in the centre part of Figure 9.3.

Note: This requires that the number of horizontal and vertical filtering steps are the same.

The second reconstruction stage shall be a repetition of the first stage, though the new 0-LL picture is now the same size as the 1-HL, 1-LH and 1-HH pictures in the center part of the figure. The result is thus the reconstructed picture, denoted 0-LL, shown on the rightmost part of Figure 9.3.

The decoding order is important because small rounding errors can accumulate if the decoding order is not correct. The vertical reconstruction shall always occur before horizontal reconstruction, i.e. all vertical lifting stages are applied first at each level. This process is defined in detail in Section 14.3.

See also informative Annex G for an illustration of the Wavelet decimation and reconstruction processes.

9.1.7.1 Wavelet Filter Support

The list of supported filters for use in the IDWT shall be as defined by the list in Table 12.1.

The inverse wavelet transform shall be implemented using a lifting process and integer arithmetic.

With the sole exception of the Haar transform, the wavelet filters require video samples that extend beyond each edge of the picture at both the horizontal and vertical axes. Edge effects shall be overcome by using an edge extension procedure that is part of each lifting process. In the case where no quantization is applied, this process ensures that the decoded picture is perfectly reproduced even at picture edges.

Note: An accessible reference to the lifting process is explained in “Ripples in Mathematics, The Discrete Wavelet Transform” as listed in the bibliography.

9.1.8 Clipping

The compression process can introduce signal errors in the video components that might result in signal overshoots and undershoots. Clipping of the output pictures (as defined in Section 14.4) shall ensure that the

signals lie within the range that can be supported by the output interface. Additional clipping may be required to ensure that the signal falls between the black and white levels defined for a specific video format or application.

10 VC-2 Stream

This section defines the structure of VC-2 streams and the overall processes for parsing and decoding. Subsequent sections define the processes for parsing and decoding individual pictures.

10.1 Pseudocode

The parsing process is normatively defined using pseudocode and/or mathematical formulae. The definitions of stream syntax operations and pseudocode shall be as defined in Section 5. The stream parsing specifications are augmented by informative parse diagrams throughout this section, each of which illustrates that part of the stream structure in graphical form.

The VC-2 stream syntax uses a state model to express the stream in a way that can be parsed and used for decoding operations (Section 5.2). The parsing and decoding operations are defined in terms of modifying the decoder state according to the data extracted from the VC-2 stream. The state of the decoder shall be stored in the global variable **`state`**. This is a map (see Section 5.4.2) and individual elements shall be accessed by means of named labels, e.g., **`state`**`[var_name]`. The state variables comprise the parameters that shall be used in parsing and decoding a picture. The variable **`state`** is a global variable and shall be accessible to all decoder functions and processes. All other variables shall be local to the function or process in which they are defined.

Some parameters are encoded in the stream as indices to tables of values. The indices shall be coded as variable length integers. This allows the tables to be extended to contain new entries, in future versions of this specification, without changing the syntax.

10.2 VC-2 Stream Syntax

A VC-2 Stream shall be a concatenation of one or more VC-2 Sequences.

The process for parsing or decoding a VC-2 Stream shall be to parse or decode all the VC-2 Sequences it contains.

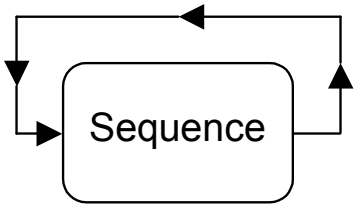

Figure 10.1 – VC-2 stream

Note: Although each VC-2 Sequence is, in effect, a newly decodable entity, care must be taken to ensure that any changes in encoded video parameters can be supported by downstream equipment. For example, changes to parameters such as frame rate and picture size might not be supported.

10.3 VC-2 Sequence Syntax

A VC-2 Sequence shall comprise an alternating sequence of parse info headers and data units. The first data unit shall be a sequence header, and further sequence headers may be inserted at any data unit point in the sequence.

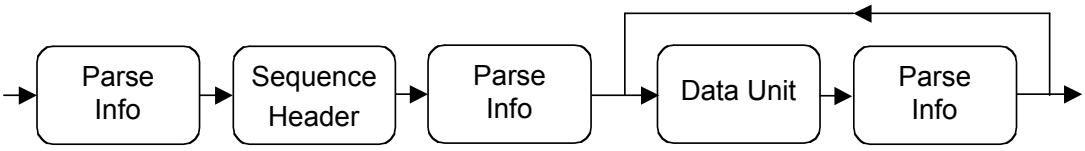

Figure 10.2 – Overview of VC-2 sequence structure

Data units may be one of the following kinds (Section 10.3.2):

- a sequence header,
- a picture,
- auxiliary data or
- padding data.

The process for parsing a VC-2 sequence shall be as defined below:

***parse_sequence*()**:	**Ref**
state = {}	
parse_info()	10.4
while(*is_end_of_sequence*() == **False**):	Table 10.2
if (*is_seq_header*() == **True**):	Table 10.2
sequence_header()	11
else if (*is_picture*() == **True**):	Table 10.2
picture_parse()	12
else if (*is_auxiliary_data*() == **True**):	Table 10.2
auxiliary_data()	10.3.2.1
else if (*is_padding*() == **True**):	Table 10.2
padding()	10.3.2.2
parse_info()	10.4

Each VC-2 Sequence shall start and end with a parse info header.

10.3.1 Parse Info Headers

Parse info headers shall contain a 32 bit code so that the decoder can be synchronized with the stream. They are defined in Section 10.4.

The parse info headers support navigating through the stream without the need to decode any data units. Each parse info header contains pointers to the location of the next and previous parse info headers within the stream. The stream can thus be thought of as a doubly linked list of data units.

Each parse info header shall contain a code that identifies the type of data held in the following data unit. This is the only information contained within the parse info headers that is needed to decode the sequence.

10.3.2 Data Units

This section describes the four types of data unit used within a sequence.

A sequence header shall contain metadata describing the coded sequence and metadata needed to decode the stream. The sequence header is defined in Section 11. The first data unit in a sequence shall be a sequence header. To support reverse-parsing applications, the last data unit in a sequence should also be a sequence header.

Note: The number of sequence headers in a VC-2 Sequence is application dependent. In a streaming application it could be appropriate to have one sequence header for each picture. When a VC-2 stream is stored in a file on a computer system it could be appropriate to have only a single sequence header per sequence.

A picture data unit shall contain sufficient data to decode a single picture (frame or field of video), subject to having parsed a sequence header within the sequence. Each picture, whether a frame or field, shall be coded with no dependency on adjacent pictures. The picture data unit is defined in Section 12.

Pictures within a sequence shall be either all frames or all fields. Where pictures are fields, a sequence shall comprise a whole number of frames (i.e., an even number of fields) and shall begin and end with a whole frame/field-pair.

If a sequence contains more than one sequence header, the data in every sequence header shall be the same (byte-for-byte identical) within the sequence.

The sequence decoding and parsing processes shall start by extracting and decoding the information contained in a sequence header. Picture data units contain the additional information needed to decode a single output picture.

Auxiliary data and padding data do not contribute to the decoding process and may be safely discarded by a decoder.

Auxiliary and padding data units comprise undefined data for the purposes of this standard. These data units (together with the correct preceding parse info header) may be interposed at any point in the stream, but may safely be skipped by a compliant decoder. Parsing auxiliary and padding data units is defined below. For the purposes of subsequent parts of this standard, the potential presence of auxiliary and padding data shall be ignored.

Padding data units shall not be used for any form of auxiliary data service or content. They may be used by an encoder, where required, to insert additional data to assist in complying with constant or constrained bit rate requirements.

10.3.2.1 Auxiliary Data

The *auxiliary_data*() process for reading auxiliary data shall be as follows:

***auxiliary data*()**:	**Ref**
byte_align()	A.1.3
for i = 1 **to** (state[next_parse_offset]-13)	10.4
read_byte()	A.1.1

10.3.2.2 Padding

The *padding*() process for reading padding data shall be as follows:

***padding*()**:	**Ref**
byte_align()	A.1.3
for i = 1 **to** (state[next_parse_offset]-13)	10.4
read_byte()	A.1.1

10.4 Parse Info Header Syntax

The parse info header provides information identifying the subsequent data unit type and length codes determining the number of bytes from the current parse info header to the next and previous parse info headers,

The parse info header shall be byte-aligned. It shall be present:

- at the beginning of a sequence,
- at the end of a sequence,
- before each data unit (whether sequence header, picture, padding or auxiliary data).

The parse info header shall consist of 13 bytes and shall be byte-aligned within the sequence. Thus it ensures that succeeding data elements are byte aligned.

The value of the parse code, which is a component of the parse info, shall be used to determine the type and format of the subsequent data structures.

The *parse_info*() process for reading parse info headers shall be as follows:

***parse_info*():**	**Comment**	**Type**
read_uint_lit(4)	Parse info prefix	UInt32
state[parse_code] = *read_byte*()	Parse code	UInt8
state[next_parse_offset] = *read_uint_lit*(4)	Next parse offset	UInt32
state[previous_parse_offset] = *read_uint_lit*(4)	Previous parse offset	UInt32

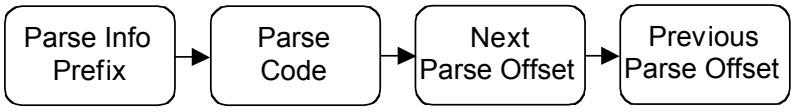

Figure 10.3 – Parse Info header syntax

The parse info parameters shall satisfy the following constraints:

- The parse info prefix shall be 0x42 0x42 0x43 0x44[1].
- The parse code shall be one of the supported values set out in Table 10.1.
- The next parse offset shall be the number of bytes from the first byte of the current parse info header to the first byte of the next parse info header, if there is one or if an encoder is able to determine the correct value at the time of encoding. Otherwise the next parse offset value shall be zero[3].

- The previous parse offset shall be the number of bytes from the first byte of the current parse info header to the first byte of the previous parse info header, if there is one. If there is no preceding parse info header, it shall be zero.

Consequently, the previous parse offset value of the current parse info header shall equal the next parse offset value of the previous parse info header, if there is one.

Notes:

1. The parse info prefix is the character string: “BBCD” as expressed by ISO/IEC 646.
2. The parse info prefix, next parse offset and previous parse offset values are provided to support navigation. They are not required to decode the sequence, but are added to the byte stream to simplify parsing. See Section 10.5.1.
3. The parse offset values will normally be non-zero. However, at the beginning and end of a stream there are no preceding or following parse info headers respectively. In these circumstances the value of the offset is zero. Typically, a zero value for a parse offset can only occur in a sequence header or an end of sequence data unit, and then only at the beginning and end of a stream. However, real-time, variable bit-rate encoders that are unable to determine the position of the next parse info header without adding significant encoding delay can set the next parse offset value to zero if the parse info header precedes a picture.

10.4.1 Parse Codes

The values of parse codes allowed within the VC-2 syntax shall be as shown in Table 10.1.

Table 10.1 – Parse Code Values

Parse Code (hex)	Binary code (informative)	Description
Generic VC-2 Syntax:		
0x00	0000 0000	Sequence header
0x10	0001 0000	End of sequence
0x20	0010 0000	Auxiliary data
0x30	0011 0000	Padding data
Core syntax:		
0x08	0000 1000	Picture (arithmetic coding)
0x48	0100 1000	Picture (no arithmetic coding)
Low delay syntax:		
0xC8	1100 1000	Picture (no arithmetic coding)

The three least significant bits (bits 0, 1, 2) in the parse code shall always be zero. Other permutations of bits 0, 1 and 2 shall be reserved for future expansion.

The parse codes shall be associated with a group of functions, listed in Table 10.2 below, which shall determine the type of subsequent data and the parsing and decoding processes which shall be used. All functions shall return a Boolean value.

Table 10.2 – Parse Code Functions

Function Name:	Decoding Operation:	Type
is_seq_header()	**return** (**state**[*parse_code*] == 0x00)	**Boolean**
is_end_of_sequence()	**return** (**state**[*parse_code*] == 0x10)	**Boolean**
is_auxiliary_data()	**return** ((**state**[*parse_code*]&0xF8) == 0x20)	**Boolean**
is_padding_data()	**return** (**state**[*parse_code*] == 0x30)	**Boolean**
is_picture()	**return** ((**state**[*parse_code*]&0x08) == 0x08)	**Boolean**
is_low_delay()	**return** ((**state**[*parse_code*]&0x88) == 0x88)	**Boolean**
is_core_syntax()	**return** ((**state**[*parse_code*]&0x88) == 0x08)	**Boolean**
using_ac()	**return** ((**state**[*parse_code*]&0x48) == 0x08)	**Boolean**

Future versions of this specification may introduce new parse codes. In order that decoders complying with this version of the specification may decode future versions of the coded stream, the decoder shall discard data units that immediately follow parse info blocks containing unknown parse codes.

10.4.1.1 Parse Code Values (Informative)

The rationale for the parse code values defined in Table 10.1 is as follows:

- The MS bit (bit 7) is used to indicate the picture syntax (core or low delay syntax). It only applies to pictures (i.e., where bit 3 is set), otherwise the value is undefined. Core syntax codes whole pictures rather than slices. Low delay syntax codes slices, not pictures.
- The second MS bit (bit 6) is used to indicate whether arithmetic coding is used and only applies to pictures (i.e., bit 3 is set). Core syntax can optionally use arithmetic coding. Low delay syntax does not use arithmetic coding. The permutation of the two MSBs which might indicate low delay syntax with arithmetic coding is reserved.
- The next three MS bits (bits 5, 4 and 3) indicate the type of data unit following the parse info unit. Bit 3 indicates whether it is a picture or non-picture data unit. Bits 5 and 4 indicate the 4 other parse codes.

10.5 VC-2 Sequence Decoding (Informative)

There is no one unique way of describing a VC-2 decoder. However the pseudocode below is illustrative code for a sample decoder. It emphasizes which parts of the decoding process generate decoded output data. Note that the potential presence of padding or auxiliary data is ignored for clarity.

decode_sequence():	Ref
state = {}	
video_parameters = {}	
decoded_pictures = {}	
parse_info()	10.4
while(*is_end_of_sequence*() == ***False***):	Table 10.2
if (*is_seq_header*() == **True**):	Table 10.2
video_parameters = *sequence_header*()	11
if(*is_picture*() == **True**):	Table 10.2
picture_parse()	12
decoded_pictures[*length*(decoded_pictures)] = *picture_decode*()	14
parse_info()	10.4
return {video_parameters, decoded_pictures}	

The *decode_sequence()* process returns a set containing the parameters (metadata) describing the decoded video and an array of decoded pictures. Each decoded picture contains a picture number and three decoded video components. This is the decoded output.

The pseudocode describes the decoding process. Decoding starts by clearing the decoder state and the decoder output. Thus video sequences can be decoded as independent entities. The first data extracted from the VC-2 stream is parse information. Parse info indicates what type of data unit follows, and this information is stored in the decoder state. The decoder continues to read pairs of data unit and parse info until the end of the sequence is reached. The end of sequence is indicated by data in the final parse info header. If a data unit is a sequence header the decoded video parameters are updated with the information contained in the header. If the data unit is a picture then the picture is decoded and appended to the end of the array of decoded output pictures.

(For clarity, this pseudocode ignores the presence of auxiliary data and padding data in the sequence. Nor does it illustrate providing the picture numbers, which are coded in the stream, nor details of the coding parameters, which are needed by some applications.)

Each picture data unit within a sequence starts with a picture header (Section 12.1) that contains a 32-bit picture number. Picture numbers within a sequence increment by one for each new picture.

10.5.1 Non Sequential Picture Decoding (Informative)

The ability to decode pictures in a non-sequential manner is important for many applications, such as video editing. Non-sequential access means decoding a stream in any manner other than decoding pictures sequentially from the beginning of the stream to the end. Non-sequential picture access is outside the scope of this specification. Nevertheless the VC-2 stream has been designed to support this feature. This section provides informative notes on this aspect of the VC-2 stream specification.

Stream navigation, including non-sequential access is supported by the information in the parse info headers in the stream. Details of parse info headers are defined in Section 10.4.

In order to start decoding, other than at the start of a stream, the decoder must first synchronize to the stream. The parse info prefix is present to support such synchronization. A decoder would first search for the parse info prefix to locate the start of a parse info header. The parse info prefix is not guaranteed to occur uniquely within parse info headers (the entropy coding used in VC-2 precludes this), the parse info prefix can, by chance, occur within a data unit. The decoder could read the next or previous parse info offsets to confirm that an occurrence of the parse info prefix corresponds to a parse info header.

When the decoder finds a parse info prefix it can skip backwards by the value of the previous parse offset and check whether the next four bytes are also equal to the parse info prefix. If so the decoder can be reasonably certain that it has found a genuine parse info prefix.

If it does not find another parse info prefix, it was probably unlucky enough to have found a spurious parse info prefix. In this case it can search for the next prefix value in the stream and repeat the test. The probability of a spurious parse info prefix is low: 1 in 2^{32}, since the prefix is 4 bytes long. The test for two independent parse info prefixes, correctly separated is, therefore, more than adequate in practice.

Having synchronized with the stream the decoder now needs to locate a sequence header in order to find parameters needed to decode pictures. This can be done by moving forward (or backward) through the stream, between successive parse info headers, using next (or previous) parse offsets, until a parse info header is found containing the parse code for a sequence header. Decoding can now start, as if from the beginning of the stream.

The VC-2 stream also supports seeking to a particular picture number. The first four bytes of the picture header contain a 4-byte picture number. Picture numbers provide a continuously incrementing identifier for each picture with a sequence. So, to find a particular picture, the decoder can move forward or backward in the stream, using the offsets in the parse info headers, until the correct picture number is reached. A picture can be decoded once the parameters within a sequence header, in the same sequence, have been read.

11 Sequence Header

This section defines the structure of the sequence header syntax. The sequence header shall be byte aligned.

Parsing the sequence header consists of reading the sequence parameters (parse parameters, base video format, source parameters and picture coding mode) and initializing the decoder parameters. The decoder parameters shall be initialized by the *set_coding_parameters* process (below).

The sequence header shall remain byte identical throughout a sequence.

The process for parsing the sequence header shall be as follows:

***sequence_header*():**	**Type**	**Ref**
byte_align()		
parse_parameters()		11.1
base_video_format = *read_uint*()	**UInt**	11.2
video_parameters = *source_parameters*(base_video_format)		11.3
picture_coding_mode = *read_uint*()	**UInt**	11.4
set_coding_parameters(video_parameters, picture_coding_mode)		11.5
return video_parameters		

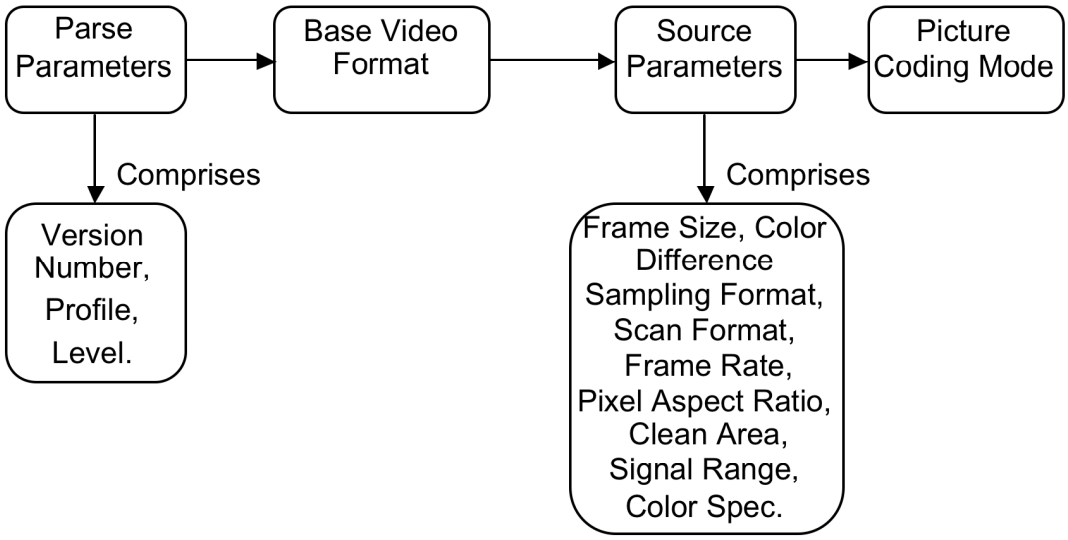
Parse Parameters
Base Video Format
Source Parameters
Picture Coding Mode
Comprises
Comprises
Version Number, Profile, Level.
Frame Size, Color Difference Sampling Format, Scan Format, Frame Rate, Pixel Aspect Ratio, Clean Area, Signal Range, Color Spec.

Figure 11.1 – Sequence Header

Parse parameters contain information that a decoder can use to determine whether it is able to parse or decode the stream. Parse parameters are not used to decode the stream.

The base video format shall be a numerical index denoting a default set of parameters that describe the video source. The numerical index values shall be as defined in Annex C.

For many common applications, these predefined formats will be sufficient without the need for further metadata to be present in the stream. However, to provide flexibility, source parameters may override some or all of the parameters indicated by the base video format.

Source parameters are parameters that describe the source video, not all of which are required to decode the stream. The source parameters are needed by applications that use the decoded video and so should be made available to them.

Picture coding mode indicates how the video frames have been coded (e.g., as a sequence of frames or fields).

Once the base video format, source parameters and picture coding mode (as illustrated in Figure 11.1) have been read from the stream, the information they contain may be decoded to provide the parameters used for decoding pictures. It is the purpose of the *set_coding_parameters* process to initialize these parameters.

11.1 Parse Parameters

This section specifies the structure of the parse parameters, which shall be defined as follows:

parse_parameters():	Comment	Ref	Type
read_uint()	major version	11.1.1	**UInt**
read_uint()	minor version	11.1.1	**UInt**
read_uint()	profile	11.1.2	**UInt**
read_uint()	level	11.1.2	**UInt**

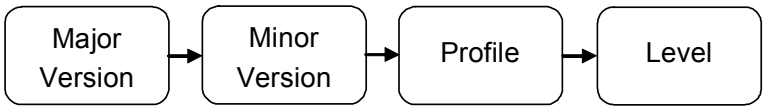

Figure 11.2 – Parse Parameters

The parse parameters data values shall be constant (byte-for-byte identical) for all instances of the sequence header within a VC-2 sequence. For VC-2 Stream interchange, the VC-2 Profile and Level values should also be identical for all VC-2 sequences within a VC-2 stream.

11.1.1 Version Number

The version number of the VC-2 syntax specification shall be used by the decoder to determine whether it can decode the sequence.

The major version number shall define the version of the syntax with which the stream complies. A decoder complies with a major version number if it can parse all bit streams that comply with that version number. Decoders that comply with a major version of the specification may not be able to parse the bit stream corresponding to a later specification.

Depending on the profile and level defined, a decoder compliant with a given major version number may still not be able to decode fully all parts of a stream.

All minor versions of the specification shall be functionally compatible with earlier minor versions with the same major version number. Later minor versions may contain corrections, clarifications, and removal of ambiguities. Later minor version numbers shall not contain new features or new normative provisions.

Functional compatibility shall imply that a decoder with the same major version number and later minor version number as a stream shall be capable of decoding the stream and producing decoded pictures that are substantially equivalent to that of a decoder with the same version number as the stream. This shall be assessed by the appropriate technology committee when the minor version number is incremented.

The major version number of a stream compliant with this version of the VC-2 standard shall be 1.

The minor version number of a stream compliant with this version of the VC-2 standard shall be 0.

11.1.2 Profiles and Levels

The profile shall define the toolset that is sufficient to decode a VC-2 sequence.

The level shall define decoder resources (picture and data buffers; computational resources) sufficient to decode a sequence.

Profiles are defined in Annex D.1. Levels are described in Annex D.2.

11.2 Base Video Format

The value of `base_video_format` decoded in parsing the sequence header shall be an index into one of a set of predefined video formats defined in Table 11.1. For each predefined video format entry in the table, the base video parameters shall be as defined in Annex C.

The selection of a base format represents an initial approximation to the video format, which can then be refined to capture all the video format characteristics accurately by overriding parameters as necessary. In particular, the predefined video formats listed in Table 11.1 do not represent all the video formats supported by VC-2; any video format parameters may in principle be defined and supported by a VC-2 sequence.

These base parameters may be modified by subsequent metadata present in the stream, with the exception of the top field first parameter which shall only be set by the base video format (see Section 11.3.4).

Table 11.1 – VC-2 base video formats

`base_video_format`	Video Format Name (Informative)
0	Custom Format
1	QSIF525
2	QCIF
3	SIF525
4	CIF
5	4SIF525
6	4CIF
7	SD 480I-60
8	SD 576I-50
9	HD 720P-60
10	HD 720P-50
11	HD 1080I-60
12	HD 1080I-50
13	HD 1080P-60
14	HD 1080P-50
15	DC 2K-24
16	DC 4K-24
17	UHDTV 4K-60
18	UHDTV 4K-50
19	UHDTV 8K-60
20	UHDTV 8K-50
21	HD 1080P-24
22	SD Pro486

Notes:

1. The custom format is intended for use when no other suitable base video format is available from the table. Video format default values will still be set as per annex C, but these are token values that will be almost wholly overridden by the subsequent source parameter values.
2. The base video format ought to be as close as possible to the desired video format, especially in terms of picture dimensions and frame rate.
3. The video format name is purely informative and does not imply adherence to any video interface.

11.3 Source Parameters

The source parameters are intended indicate the format of the video that was originally encoded, and they provide metadata that indicates how the decoded video should be displayed.

The source parameters shall comprise frame size, sampling format, scan format, frame rate, aspect ratio, clean area, signal range and color specification. The frame size, sampling format, scanning format and the signal range are required to decode the video. Display and downstream processing falls outside the scope of this specification, hence the interpretation of the other parameters (not required to decode the video) is not normatively defined, with the exception of frame rate (Section 11.3.5). The frame rate imposes requirements on compliant decoders for a given level and profile (Annex D).

Source parameter data shall remain constant throughout a VC-2 sequence.

Default values for the source parameters shall be derived from the base video format, as defined in Annex C. These default values shall be the source parameters unless they are overridden with alternative values encoded as part of the source parameters part of the stream.

The *source_parameters*() process shall return a structure defining the video source parameters (video_parameters). It shall be defined as follows:

***source_parameters*(video_format):**	**Ref**
video_parameters = *set_source_defaults*(base_video_format)	11.3.1
frame_size(video_parameters)	11.3.2
color_diff_sampling_format(video_parameters)	11.3.3
scan_format(video_parameters)	11.3.4
frame_rate(video_parameters)	11.3.5
pixel_aspect_ratio(video_parameters)	11.3.6
clean_area(video_parameters)	11.3.7
signal_range(video_parameters)	11.3.8
color_spec(video_parameters)	11.3.9
return video_parameters	

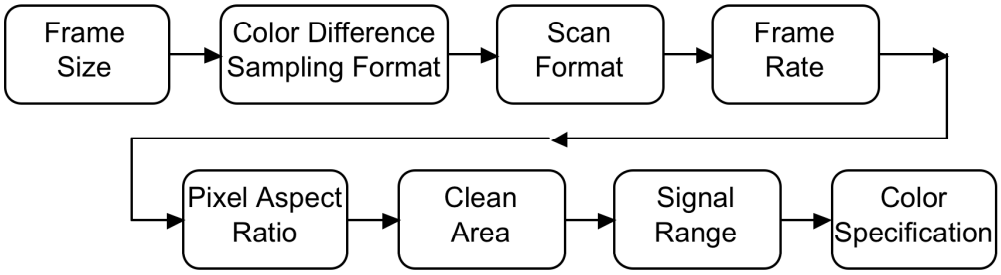

Figure 11.3 – Source Parameters

11.3.1 Setting Source Defaults

The function that sets the default values of the source video parameters shall take the base video format index as an argument. That is, the signature of this function is: *set_source_defaults*(base_video_format) where base_video_format is an unsigned integer. The function returns a map of source video parameters.

The source video parameters shall be set, based on the base video format index, as defined in Annex C. The parameters set by this function shall be: frame size, sampling format (4:4:4, 4:2:2 or 4:2:0), scan format (progressive or interlace), frame rate, aspect ratio, clean area, signal range, color specification. The labels used to access the map returned by the function shall be as defined in the subsequent sections that specify how to override the default video source parameters.

Note: These sections each specify a boolean flag that determines if custom video parameters are required. If the custom parameters are required, then the custom parameters are present in the stream else the custom parameters are omitted from the stream. For example, if video_format == 4, CIF defaults are set, with picture size equal to 352x288 with a 4:2:0 color difference format, amongst other parameters. A frame width of 360 pixels can be encoded by setting the boolean flag to True and overriding the default CIF frame dimensions.

11.3.2 Frame Size

The frame size decoding process shall be defined as follows:

***frame_size*(video_parameters):**	**Type**
custom_dimensions_flag = *read_bool()*	**Bool**
if(custom_dimensions_flag == **True**)	
video_parameters[frame_width] = *read_uint*()	**UInt**
video_parameters[frame_height] = *read_uint*()	**UInt**

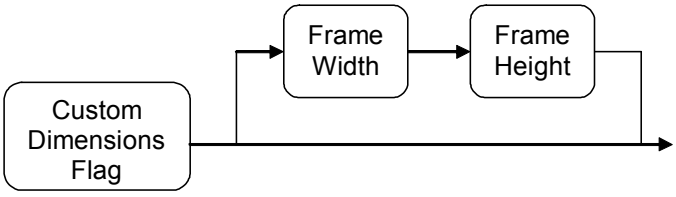

Figure 11.4 – Frame Size

Thus if custom_dimensions_flag is set to **True**, the frame size defined by the default values shall be overridden by the new values.

The frame width shall correspond to the width of the coded video, in pixels, that is coded in the stream. The frame height shall correspond to the number of lines per frame in the coded video, irrespective of whether the coded video is progressively scanned or is interlaced.

11.3.3 Color difference Sampling Format

The color difference sampling format decoding process shall be defined as follows:

***color_diff_sampling_format*(video_parameters):**	**Type**
custom_color_diff_format_flag = *read_bool()*	**Boolean**
if(custom_color_diff_format_flag == **True**)	
video_parameters[color_diff_format_index] = *read_uint*()	**UInt**

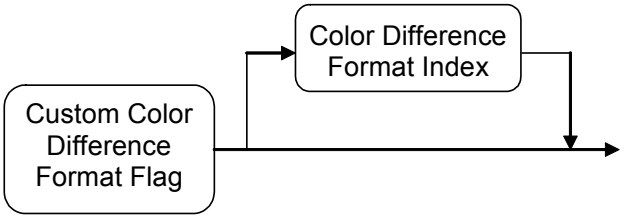

Figure 11.5 – Sampling Format

Thus if the custom_color_diff_format_flag is set to **True**, the color difference sampling format defined by the default value shall be overridden by the new values defined by the index value.

The decoded value of video_params[color_diff_format_index] shall lie in the range 0 to 2 with values defined as in Table 11.2.

Table 11.2 – Color difference sampling formats

Index	Color difference format
0	4:4:4
1	4:2:2
2	4:2:0

The color difference sampling format shall be used to determine the width and height of the color difference components of the coded video, as described in Section 11.5.1.

11.3.4 Scan Format

The scan format parameter shall indicate whether the source video represents progressive frames or interlaced fields.

The scan format decoding process shall be defined as follows:

***scan_format*(video_parameters):**	**Type**
custom_scan_format_flag = *read_bool()*	**Boolean**
if(custom_scan_format_flag == **True**):	
video_parameters[source_sampling] = *read_uint()*	**UInt**

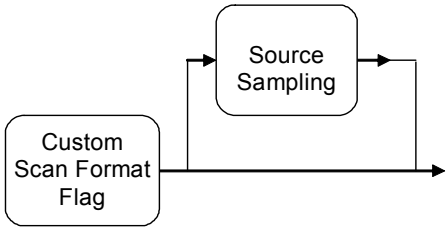

Figure 11.6 – Scan format

If the custom_scan_format_flag is set to **True**, the scan format parameters defined by the default values shall be overridden by the new values.

If video_parameters[source_sampling] is set to 0, then the source video shall be progressively sampled. If it is 1, then the source video shall be interlaced. Values greater than 1 shall be reserved.

The parameter video_parameters[top_field_first] shall be **True** if the top line of the frame is in the earlier field, else video_parameters[top_field_first] shall be **False**. This shall be set only by the base video format and cannot be overridden in the source parameters.

Both interlaced and progressive video may be coded as fields or frames.

11.3.5 Frame Rate

The frame rate value (in frames per second) shall be defined by the ratio of the parameters:

video_parameters[frame_rate_numer] divided by video_parameters[frame_rate_denom]

The process for decoding the frame rate parameters shall be as follows:

***frame_rate*(video_parameters):**	**Type**
custom_frame_rate_flag = *read_bool*()	**Boolean**
if(custom_frame_rate_flag == **True**):	
index = *read_uint()*	**UInt**
if(index == 0):	
video_parameters[frame_rate_numer] = *read_uint()*	**UInt**
video_parameters[frame_rate_denom] = *read_uint()*	**UInt**
else:	
preset_frame_rate(video_parameters, index)	

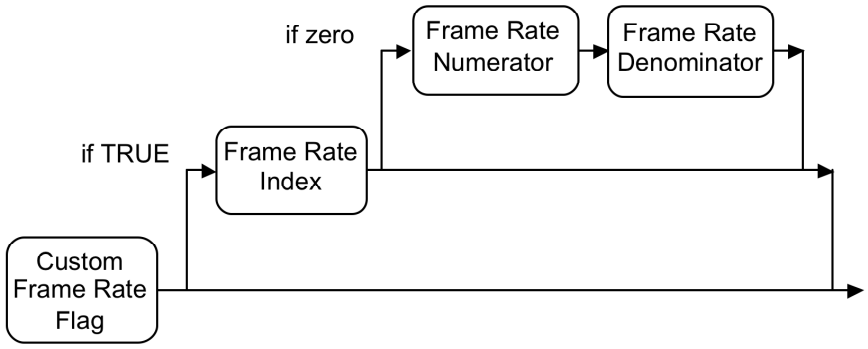

Figure 11.7 – Frame rate

If `custom_frame_rate_flag` is set to **`True`**, the frame rate defined by the default values shall be overridden by the new values defined by the index value.

Note: What is encoded is the frame rate and not picture rate. If the source video is interlaced, and is encoded as fields rather than frames, then the picture rate is twice the encoded frame rate.

The value of `index` shall lie in the range from 0 to the maximum value defined by Table 11.3.

If the value of `index` is 0, then the frame rate numerator and denominator shall be individually defined by unsigned integer values.

For values >0, the process `preset_frame_rate(video_parameters, index)` shall set the frame rate elements in `video_parameters` according to Table 11.3.

Table 11.3 – Preset frame rate values

Frame Rate Index	Frame Rate Numerator	Frame Rate Denominator	Frame Rate (Hz)
1	24000	1001	24000/1001
2	24	1	24
3	25	1	25
4	30000	1001	30000/1001
5	30	1	30
6	50	1	50
7	60000	1001	60000/1001
8	60	1	60
9	15000	1001	15000/1001
10	25	2	25/2
11	48	1	48

11.3.6 Pixel Aspect Ratio

The pixel aspect ratio shall be defined as the ratio of the parameters:

`video_parameters[pixel_aspect_ratio_numer]:video_parameters[pixel_aspect_ratio_denom]`

The process for decoding the pixel aspect ratio parameters shall be defined as follows:

***aspect_ratio*(video_parameters):**	**Type**
custom_pixel_aspect_ratio_flag = *read_bool*()	**Boolean**
if(custom_pixel_aspect_ratio_flag == **True**):	
index = *read_uint()*	**UInt**
if(index == 0):	
video_parameters[pixel_aspect_ratio_numer] = *read_uint()*	**UInt**
video_parameters[pixel_aspect_ratio_denom] = *read_uint()*	**UInt**
else:	
preset_aspect_ratio(video_parameters, index)	

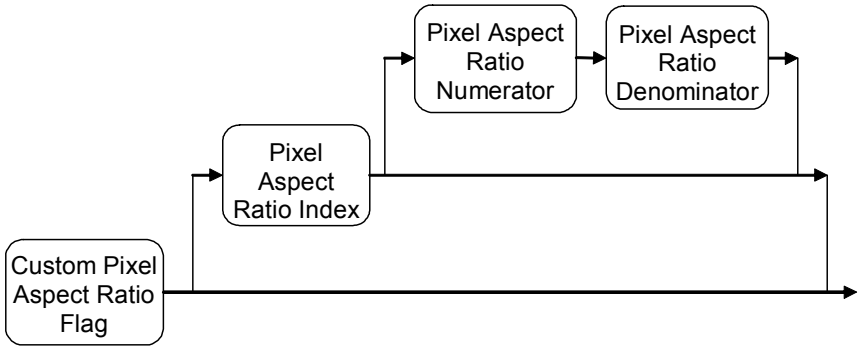

Figure 11.8 – Pixel Aspect Ratio

If `custom_pixel_aspect_ratio_flag` is set to **`True`**, the pixel aspect ratio defined by the default values shall be overridden by the new values defined by the index value.

The value of `index` shall lie in the range 0 to 6.

If the value is 0, then the pixel aspect ratio numerator and denominator shall be individually defined by unsigned integer values.

For values >0, the process *`preset_aspect_ratio`*`(video_parameters, index)` shall set the pixel aspect ratio elements in `video_parameters` according to Table 11.4.

Table 11.4 – Preset aspect ratio values

Aspect Ratio Index	Pixel Aspect Ratio
1	1:1
2	10:11 (4:3 525 line systems)
3	12:11 (4:3 625 line systems)
4	40:33 (16:9 525 line systems)
5	16:11 (16:9 625 line systems)
6	4:3 (reduced horizontal resolution systems)

Notes:

1. The pixel aspect ratio value defines the intended ratio of the pixel sampling such that the viewed picture has no geometric distortion. The pixel aspect ratio of an image is the ratio of the spacing of horizontal samples (pixels) to the spacing of vertical samples (picture lines) on the display device. Pixel aspect ratios (PARs) are fundamental properties of sampled images because they determine the displayed shape of objects in the image. Failure to use the right PAR will result in distorted images, for example circles will be displayed as ellipses etc.
2. The pixel aspect ratios shown in Table 11.4 assume a 704 x 480 active picture for the 525-line systems and a 704 x 576 active picture for the 625-line systems. See also Annex F4.1.1.
3. Some video processing tools require an image aspect ratio. This can be derived from the pixel aspect ratio by multiplying the ratio of horizontal to vertical pixels by the pixel aspect ratio. So, for example, for a 704 x 480 line picture, with a pixel aspect ratio of 10:11 the image aspect ratio is (704 x 10)/(480 x 11) which is exactly 4:3.

11.3.7 Clean Area

The process for decoding the clean area parameters shall be as follows:

***clean_area*(video_parameters):**	**Type**
custom_clean_area_flag = *read_bool*()	**Boolean**
if(custom_clean_area_flag == **True**):	
video_parameters[clean_width] = *read_uint*()	**UInt**
video_parameters[clean_height] = *read_uint*()	**UInt**
video_parameters[left_offset] = *read_uint*()	**UInt**
video_parameters[top_offset] = *read_uint*()	**UInt**

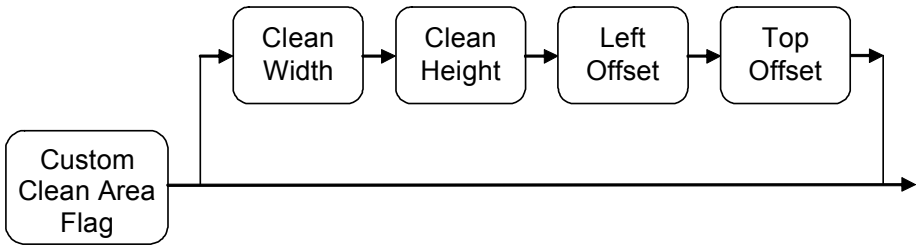

Figure 11.9 – Clean area

If custom_clean_area_flag is set to **True**, the clean area parameters defined by the default values shall be overridden by the new values.

The following restrictions shall apply:

```
clean_width + left_offset ≤ video_parameters[frame_width]

clean_height + top_offset ≤ video_parameters[frame_height]
```

Note: The meaning and use of clean area is application defined: it might correspond to that part of the picture which is to be displayed, or define a “container” within a picture of larger size.

11.3.8 Signal Range

The signal range parameters indicate how the signal range of the picture component data, decoded by the VC-2 decoder, should be adjusted prior to the color matrixing operations (described in informative Annex F).

The signal range parameters shall also be used to determine the luma depth and color difference depth parameters (Section 11.5.2) and the resulting clipping levels applied to the decoded video (Section 14.4).

The process for decoding the signal range parameters shall be as follows:

signal_range(video_parameters):	**Type**
custom_signal_range_flag = *read_bool*()	**Boolean**
if(custom_signal_range_flag == **True**):	
index = *read_uint()*	**UInt**
if(index == 0):	
video_parameters[luma_offset] = *read_uint()*	**UInt**
video_parameters[luma_excursion] = *read_uint()*	**UInt**
video_parameters[color_diff_offset] = *read_uint()*	**UInt**
video_parameters[color_diff_excursion] = *read_uint()*	**UInt**
else:	
preset_signal_range(video_parameters, index)	

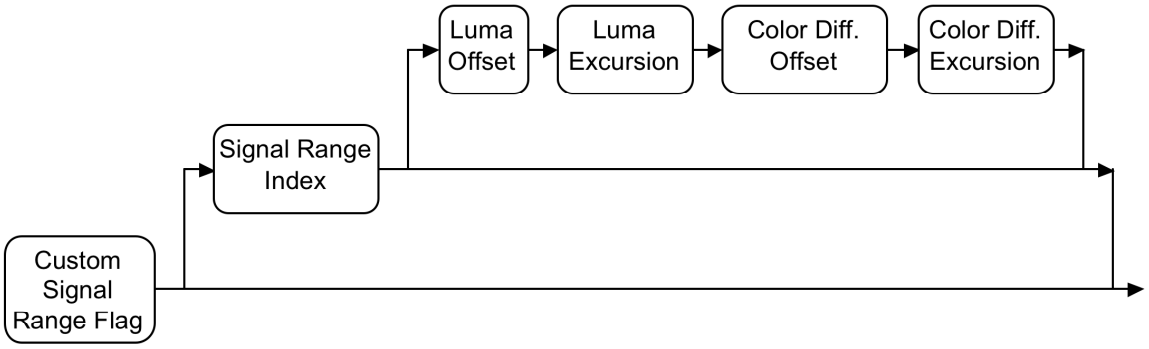

Figure 11.10 – Signal range

If custom_signal_range_flag is set to **True**, the signal ranges defined by the default values shall be overridden by the new values defined by the index value.

The value of index shall lie in the range 0 to 4.

If the index value is 0, then the luma and color difference offset and excursion values are individually defined as per the table above.

VC-2 bitstreams that contain a signal range that is not a preset value, should specify a custom signal range (or ranges) within the applicable VC-2 level specification.

For index values >0, the process *preset_signal_range*(video_parameters, index) shall set the signal range elements in video_parameters according to Table 11.5.

Table 11.5 – Preset signal ranges

index	Description	luma_offset	luma_excursion	color_diff_offset	color_diff_excursion
1	8-bit Full Range	0	255	128	255
2	8-bit Video	16	219	128	224
3	10-bit Video	64	876	512	896
4	12-bit Video	256	3504	2048	3584

Note: Decoded video is represented within the decoder specification as bi-polar signals. An offset is added when video is output so that it is represented by unsigned integer values.

11.3.9 Color Specification

The color specification shall consist of three component parts:

- Color primaries
- Color matrix
- Transfer function

Defaults are available for all three parts collectively and individually.

The process for decoding the color specification parameters shall be follows:

color_spec(video_parameters):	**Type**	**Ref**
custom_color_spec_flag = *read_bool()*	**Boolean**	
if(custom_color_spec_flag == **True**):		
index = *read_uint()*	**UInt**	
*preset_color_spec(*video_parameters, *index)*		
if(index == 0):		
color_primaries(video_parameters)		11.3.9.1
color_matrix(video_parameters)		11.3.9.2
transfer_function(video_parameters)		11.3.9.3

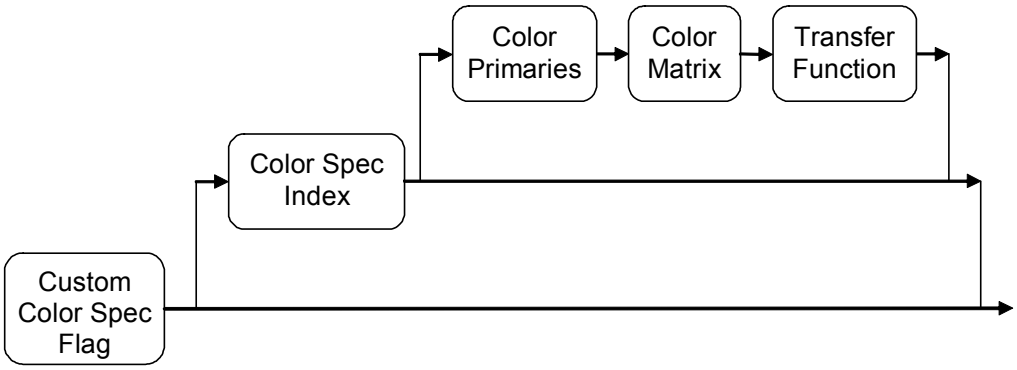

Figure 11.11 – Color specification syntax

If `custom_color_spec_flag` is set to **`True`**, the color specification defined by the default values shall be overridden by the new values defined by the index value.

The value of `index` shall lie in the range 0 to 4.

The process *`preset_color_spec`*`(video_parameters, index)` shall set the color specification elements (that is color primaries, color matrix and transfer function) in `video_parameters` to be as defined in Table 11.6.

Table 11.6 – Preset color specifications

Index	Description	Color Primaries	Color Matrix	Transfer Function
0	Custom	HDTV	HDTV	TV Gamma
1	SDTV 525	SDTV 525	SDTV	TV Gamma
2	SDTV 625	SDTV 625	SDTV	TV Gamma
3	HDTV	HDTV	HDTV	TV Gamma
4	D-Cinema	D-Cinema	Reversible	D-Cinema

If the value of `index` is 0, then the color primaries, color matrix and transfer function values may additionally be overridden individually according to the following three sub-sections.

Note: The names used for describing the color primaries, color matrix and transfer function values are defined in the tables of Sections 11.3.9.1, 11.3.9.2 and 11.3.9.3.

11.3.9.1 Color Primaries

The color primaries decoding process shall be defined as follows:

`color_primaries``(video_parameters):`	**`Type`**
`custom_color_primaries_flag = `*`read_bool()`*	**`Boolean`**
`if``(custom_color_primaries_flag == `**`True`**`):`	
`index = `*`read_uint()`*	**`UInt`**
`preset_color_primaries``(video_parameters,index)`	

If `custom_color_primaries_flag` is set to **`True`**, the color primaries defined by the default values shall

be overridden by the new values defined by the index value.

The value of `index` shall lie in the range 0 to 3.

The process *`preset_color_primaries`*`(video_parameters, index)` shall set the color primaries element in `video_parameters` according to Table 11.7.

Table 11.7 – Preset color primaries

Index	Description	Specification	Comment
0	HDTV	ITU-R BT.709	Also Computer, Web, sRGB
1	SDTV 525	ITU-R BT.601	525 Primaries
2	SDTV 625	ITU-R BT.601	625 Primaries
3	D-Cinema	SMPTE 428.1	CIE XYZ

11.3.9.2 Color Matrix

The color matrix decoding process shall be defined as follows:

`color_matrix``(video_parameters):`	**`Type`**
`custom_color_matrix_flag = read_bool()`	**`Boolean`**
`if``(custom_color_matrix_flag == True):`	
`index = read_uint()`	**`UInt`**
`preset_color_matrix``(video_parameters, index)`	

If `custom_color_matrix_flag` is set to **`True`**, the color matrix defined by the default values shall be overridden by the new values defined by the index value.

The value of `index` shall lie in the range 0 to 2. The process *`preset_color_matrix`*`(video_parameters, index)` shall set the color matrix element in `video_parameters` according to Table 11.8.

Table 11.8 – Preset color matrices

Index	Description	Specification	Color Matrix	Comment (informative)
0	HDTV	ITU-R BT.709	K_R= 0:2126,K_B= 0:0722	Also Computer and Web
1	SDTV	ITU-R BT.601	K_R= 0:299,K_B= 0:114	
2	Reversible	ITU-T H.264	YCoCg	

11.3.9.3 Transfer Function

The transfer function decoding process shall be defined as follows:

transfer_function(video_parameters):	**Type**
custom_transfer_function_flag = *read_bool()*	**Boolean**
if(custom_transfer_function_flag == **True**):	
index = *read_uint()*	**UInt**
preset_transfer_function(video_parameters ,index)	

If custom_transfer_function_flag is set to **True**, the transfer function defined by the default values shall be overridden by the new values defined by the index value.

The value of index shall lie in the range 0 to 3.

The process *preset_transfer_function*(video_parameters, index) shall set the transfer function element in video_parameters according to Table 11.9.

Table 11.9 – Preset transfer functions

index	Description	Specification
0	TV Gamma	ITU-R BT.709
1	Extended Gamut	ITU-R BT.1361 (extended color gamut)
2	Linear	Linear
3	D-Cinema Transfer Function	SMPTE 428.1

11.4 Picture Coding Mode

The picture coding mode part of the sequence header shall determine whether source video is coded as frames or fields.

If the picture coding mode is 1 then pictures shall correspond to fields; if the picture coding mode is 0, then pictures shall correspond to frames. Values greater than 1 shall be reserved.

If video is coded as fields then the earliest field in each frame shall have an even picture number (Section 12.1). That is, the LSB of the frame number, expressed as a binary number, indicates field parity.

With field coding, each frame shall be split into two fields as indicated by the scan format (Section 11.3.4).

An effect of field coding shall be to halve the vertical dimensions of coded pictures. Hence, once the picture coding mode is known, the picture dimensions shall be set and stored as part of the global **state** variable.

Note: It is possible to code progressive video as fields. In this case, the assignment of frame lines to fields will be determined by the value of top_field_first in the base video format parameters (Annex C). Note that, according to Section 11.3.4, this base format default cannot be overridden for progressive video, as to do so would be artificial. Sometimes progressive source video is conveyed as if it were interlaced (for example using interlaced SDI modes), and could be signaled as such. This is known as progressive segmented frames. A VC-2 encoder could detect progressive segmented frames, and signal video as progressive, yet still code the video as fields in order to introduce no additional buffering delay in the signal chain. Or it could take the signaled video format at face value.

11.5 Initializing Coding Parameters

The *set_coding_parameters*() process shall initialize the dimensions of the coded picture and the video depth (the maximum number of bits in a decoded video sample), which are needed to decode pictures. Picture dimensions and video depth shall remain constant throughout a VC-2 sequence and shall be

initialized as elements of the decoder **state**.

Initialization of the coding parameters shall be defined as follows:

***set_coding_parameters*(video_parameters, picture_coding_mode):**	**Ref**
picture_dimensions(video_parameters, picture_coding_mode)	11.5.1
video_depth(video_parameters)	11.5.2

11.5.1 Picture Dimensions

The *picture_dimensions()* process, which determines the size of coded pictures, shall be defined by:

```
picture_dimensions(video_parameters, picture_coding_mode):
    state[luma_width] = video_parameters[frame_width]
    state[luma_height] = video_parameters[frame_height]
    state[color_diff_width] = state[luma_width]
    state[color_diff_height] = state[luma_height]
    color_diff_format_index = video_parameters[color_diff_format_index]
    if (color_diff_format_index == 1):
        state[color_diff_width] //= 2
    if (color_diff_format_index == 2):
        state[color_diff_width] //= 2
        state[color_diff_height] //= 2
    if (picture_coding_mode == 1):
        state[luma_height] //= 2
        state[color_diff_height] //= 2
```

Note: The parameter `frame_height` in `video_parameters` refers to the height of a frame. The parameter `luma_height` in **state** refers to the height of a picture. A picture will be either a frame or a field depending on whether it is being coded in an interlaced or progressive mode.

The value of `frame_height` shall be an integer multiple of `color_diff_height`.

11.5.2 Video Depth

The video depth process, which determines the maximum number of bits required to represent a sample of the decoded video, shall be defined as follows:

```
video_depth(video_parameters):
    state[luma_depth] = intlog2(video_parameters[luma_excursion]+1)
    state[color_diff_depth] = intlog2(video_parameters[color_diff_excursion]+1)
```

VC-2 bitstreams that contain a signal range that is not a preset value should specify the video depth value within a VC-2 level specification.

Note: For the YC_GC_O format, the luma and color difference depths will be different.

12 Picture Syntax

This section defines the syntax of VC-2 picture data units.

Picture data may be successfully parsed after parsing a sequence header within the same VC-2 sequence. The picture parsing process shall be defined as follows:

***picture_parse*():**	**Ref**
byte_align()	
picture_header()	12.1
byte_align()	
wavelet_transform()	12.2

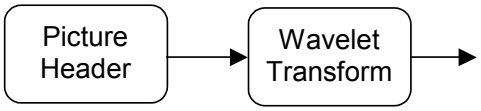

Figure 12.1 – VC-2 picture

12.1 Picture Header

The picture header shall immediately follow a parse info header that has a picture parse code (section 10.4.1). The picture header parsing process shall be defined as follows:

***picture_header*():**	**Type**
state[picture_number] = *read_uint_lit(4)*	**UInt32**

Picture numbers within a sequence shall increment by one for each successive picture. Decoders shall not rely on a continuously incrementing number across sequence boundaries.

If the video is coded as fields then the earliest field of each frame shall have an even picture number.

Note: Various operations, such as editing, can result in a discontinuity of picture number values between sequences within a VC-2 stream. Although encoders might reasonably be expected to create a stream with incrementing picture number values, a byte-stream could be edited and spliced, thus breaking up the original sequence of incrementing values. In this case, either a new sequence ought to be started within the stream or picture numbers rationalized.

12.2 Wavelet Transform

The wavelet transform syntax shall provide metadata determining the wavelet transform parameters (including filter type, transform depth, and codeblock or slice structures) together with the transformed wavelet coefficients.

The wavelet transform process for parsing transform metadata and coefficients shall be defined as follows:

***wavelet_transform*():**	**Reference**
transform_parameters()	12.3
byte_align()	
transform_data()	12

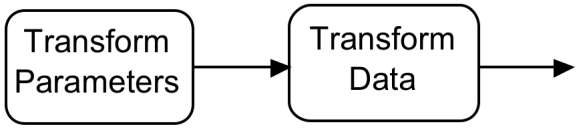

Figure 12.2 – Wavelet transform data

Parsing (unpacking) the wavelet transform data shall be as defined in Section 13.

Decoding the transformed wavelet transform data to produce decoded pictures shall be as defined in Section 14.

12.3 Transform Parameters

The wavelet transform parameters shall define the metadata required to configure the inverse wavelet transform for both the low delay and core syntax. The *transform_parameters()* process shall be as follows:

***transform parameters*():**	**Reference**
state[wavelet_index] = *read_uint()*	12.3.1
state[dwt_depth] = *read_uint()*	12.3.2
if(*is_low_delay*() == **False**)**:**	10.4.1
codeblock_parameters()	12.3.3
else:	
slice_parameters()	12.3.4.1
quant_matrix()	12.3.4.2

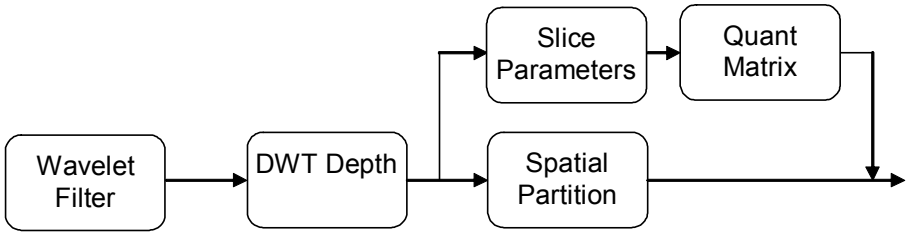

Figure 12.3 – Transform parameters

12.3.1 Wavelet Filter

The wavelet filter parameter shall define the wavelet filter used by the VC-2 stream.

The value of **state**[wavelet_index] shall lie in the range 0 to 6 with values as defined in Table 12.1.

Table 12.1 – VC-2 wavelet filters

state[wavelet_index]	**Wavelet Filter Type**
0	Deslauriers-Dubuc (9,7)
1	LeGall (5,3)
2	Deslauriers-Dubuc (13,7)
3	Haar with no shift
4	Haar with single shift per level
5	Fidelity filter
6	Daubechies (9,7) integer approximation

The implementation of the chosen wavelet filter shall be as defined in Sections 14.3.2 and 14.3.3.

Note: For consistency, the filter nomenclature (*m, n*) refers to the length of the analysis low-pass and high-pass filters in the conventional pre-filtering (i.e., before sub-sampling) model of wavelet filtering (see Annex G). They do not reflect the length of lifting filters, which operate in the sub-sampled domain: see Section 14.3. Deslauriers-Dubuc filters are normally referred to in terms of the number of vanishing moments of their synthesis filters, so the (9,7) and (13,7) filters are referred to in the literature as (2,2) and (4,2) filters, respectively.

12.3.2 Transform Depth

The transform depth parameter **state**[dwt_depth] shall determine the number of stages in the wavelet transform.

Note: A given transform depth value defines the number of subbands and thus the dimensions of the subband data array (Section 13.1).

12.3.3 Codeblock Parameters (Core Syntax Only)

In the core syntax only, each subband may be partitioned into a number of codeblocks.

The *codeblock_parameters()* process for extracting codeblock parameters shall be as defined below:

***codeblock_parameters*():**	**Type**
state[codeblock_mode] = 0	
spatial_partition_flag = *read_bool()*	**Boolean**
if(spatial_partition_flag == **False**):	
for level=0 **to** **state**[dwt_depth]:	
state[codeblocks_x][level] = 1	
state[codeblocks_y][level] = 1	
else:	
for level=0 **to** **state**[dwt_depth]:	
state[codeblocks_x][level] = *read_uint()*	**UInt**
state[codeblocks_y][level] = *read_uint()*	**UInt**
state[codeblock_mode] = *read_uint()*	**UInt**

The presence of codeblocks in subbands shall be indicated by the value of `spatial_partition_flag`.

The number of codeblocks to be used for subbands at each transform depth level shall be encoded in **state**[codeblocks_y][level] and **state**[codeblocks_x][level] for vertical and horizontal axes respectively.

The codeblock mode is encoded in **state**[codeblock_mode], which shall have value 0 or 1, with meanings as defined in Table 12.2.

Table 12.2 – Codeblock modes for spatially partitioned transform data

state[code_block_mode]	**Description**
0	Single quantizer per subband, used for all codeblocks
1	Multiple quantizers per subband, one for each codeblock

The operation of subband codeblock decoding shall be as defined in Section 12.4.3.

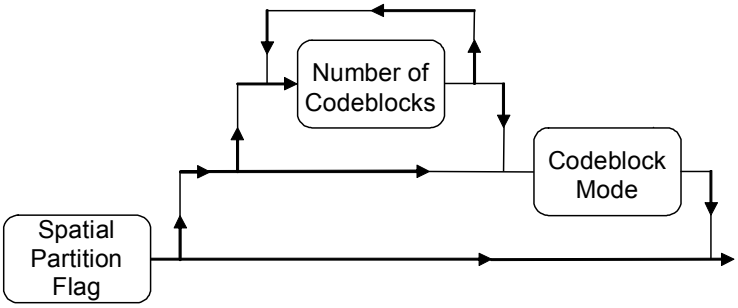

Figure 12.4 – Spatial partitioning (core syntax only)

12.3.4 Slice Coding Parameters (Low Delay Syntax Only)

This section defines the number of slices (horizontally and vertically), slice size and quantization matrix for VC-2 slices. Slices shall only be defined in the low delay syntax.

12.3.4.1 Slice Parameters

The *slice_parameters()* process for extracting slice parameter values shall be defined as follows:

***slice_parameters*():**	**Type**
state[slices_x] = *read_uint()*	**UInt**
state[slices_y] = *read_uint()*	**Uint**
state[slice_bytes_numerator] = *read_uint()*	**UInt**
state[slice_bytes_denominator] = *read_uint()*	**UInt**

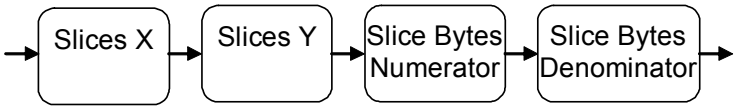

Figure 12.5 – Slice parameters

Note: The sizes of the slice parameters are unbounded. These values will be constrained by other documents that define additional VC-2 levels.

12.3.4.2 Quantization Matrices

The quantization matrix shall be used to modify the slice quantizer for each subband in a slice. The quantization matrix shall be encoded in the **state**[quant_matrix] decoder variable.

The *quant_matrix()* process for extracting quantization matrix values shall be defined as follows:

***quant_matrix*():**	**Type**	**Ref**
custom_quant_matrix = *read_bool()*	**Boolean**	
if(custom_quant_matrix == **True**):		
state[quant_matrix][0][LL] = *read_uint()*	**UInt**	
for(level = 1 to **state**[dwt_depth]:		
state[quant_matrix][level][HL] = *read_uint()*	**UInt**	
state[quant_matrix][level][LH] = *read_uint()*	**UInt**	
state[quant_matrix][level][HH] = *read_uint()*	**UInt**	
else:		
set_quant_matrix()		Annex E

If **state**[*dwt_depth*] > 4 then *custom_quant_matrix* shall be set to **True**.

If **state**[*dwt_depth*] <= 4, then *custom_quant_matrix* may be set to **True**, for example to apply a different degree of perceptual weighting (see annex E).

The function *set_quant_matrix*() shall set the quantization matrix based on the wavelet filter as per annex E. These are "unweighted" matrices, whose values merely compensate for the differential power gain of the different subband filters. For perceptual weighting a custom quantization matrix shall be used.

13 Transform Data Syntax

This section defines the process for unpacking (parsing, entropy decoding and inverse quantizing) wavelet transform coefficient data from the VC-2 stream. Wavelet coefficients shall be signed integer values and shall be extracted using the integer VLC and (optionally) arithmetic decoding functions listed in Section 9.1.2.1 and defined in Annex A. The use of arithmetic coding within a picture shall be as signaled by the parse codes defined in Section 10.4.1. The result of this process shall be a set of fully populated wavelet subband data arrays, as defined in Section 13.1.

Wavelet coefficients shall be packed in the bitstream in one of two possible formats:

1. In the core syntax, coefficients shall be grouped within individual subbands, representing a range of spatial frequencies, from the lowest to the highest. A full set of subbands shall be encoded for each video component in turn.
2. In the low delay syntax, coefficients shall be grouped into slices that represent coefficients pertaining to an area of the picture. Each slice shall contain data for all video components and spatial frequency bands. Unpacking a slice allows an area of picture to be extracted without extracting (or even receiving) the remaining picture data.

The overall process for unpacking transform data shall be as follows:

***transform_data*():**	**Ref**
if (*is_low_delay*() == **False**):	
state[y_transform] = *core_transform_data*(Y)	13.4.1
state[c1_transform] = *core_transform_data*(C1)	13.4.1
state[c2_transform] = *core_transform_data*(C2)	13.4.1
else:	
low_delay_transform_data()	13.5.1

Unpacked wavelet coefficient data shall be stored in the state variables: **state**[y_transform], **state**[c1_transform] and **state**[c2_transform] for the IDWT process (Section 14.2).

13.1 Subband Data Structure

Subband data shall be ordered by level (0, 1, 2, 3 etc) and orientation (LL, HL, LH and HH). In level 0, only the LL orientation shall be available (known also as the DC band); in the other levels only the HL, LH and HH orientations shall be available.

The 0-LL subband shall be presented first in each component (for core syntax coding) or each slice (for low delay coding). Within each subband depth level, the orientation order shall be x-HL, x-LH, x-HH (where 'x' is the transform depth level).

The subbands shall partition the spatial frequencies by orientation and level so that a four-level subband array is as illustrated in Figure 13.1.

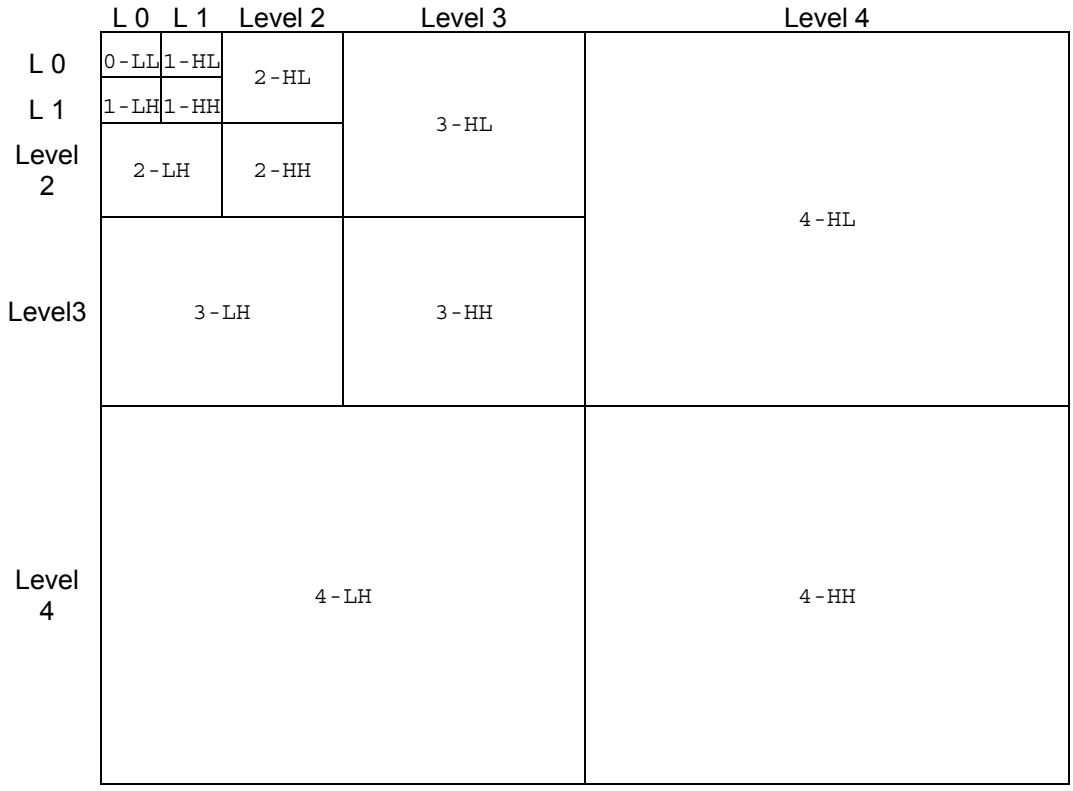

Figure 13.1 – 4-Level subband array

Note: The coefficient data is effectively a four dimensional array comprising the two dimensions *level* and *orientation* for a given subband and a further two dimensions for the H and V coefficient array within each subband.

13.1.1 Wavelet Data Initialization

This section defines the *initialize_wavelet_data(*comp*)* process, which returns a structure ready to contain the wavelet coefficients for each picture component indicated by comp.

The coefficient data shall comprise the four dimensional array coeff_data, where individual subbands shall be two-dimensional arrays accessed by level level and orientation orient: e.g.

```
band = coeff_data[level][orient]
```

Valid levels shall be integer values in the range 0 to **state**[dwt_depth] inclusive.

Level 0 shall consist of a single subband with orientation LL. All other levels shall consist of 3 subbands with orientation HL, LH, and HH in that order within the VC-2 stream.

Each subband array shall be initialized so that:

```
width(coeff_data[level][orient]) = subband_width(level, comp)
height(coeff_data[level][orient]) = subband_height(level, comp)
```

as defined in Section 13.1.2. These dimensions shall correspond to a wavelet transform being performed on a copy of the component data which has been padded (if necessary) so that its dimensions are a multiple of

$2^{\mathbf{state}[dwt\_depth]}$.

Individual subband coefficients shall be signed integers accessed by vertical and horizontal coordinates within the subband, e.g.:

```
c = coeff_data[level][orient][y][x]
```

for coordinates (x, y) with:

```
0 <= x < subband_width(level, comp) and
0 <= y < subband_height(level, comp).
```

13.1.2 Wavelet Subband Dimensions

This section defines the values of the *subband_width*(*level, comp*) and *subband_height*(*level, comp*) coefficients, giving the width and height of subbands at a given level for a given component, and hence the range of subband vertical and horizontal indices.

If comp is Y, set:

```
w = state[luma_width]
h = state[luma_height]
```

Else for color difference components, set:

```
w = state[color_diff_width]
h = state[color_diff_height]
```

The padded dimensions of the component shall be defined by:

$$scale = 2^{\mathbf{state}[dwt\_depth]}$$

$$pw = scale * ((w+scale-1) //scale)$$

$$ph = scale * ((h+scale-1) //scale)$$

The subband width and subband height values shall be defined by:

- if level = 0:

$$subband\_width(level, comp) = pw // 2^{\mathbf{state}[dwt\_depth]}$$

$$subband\_height(level, comp) = ph // 2^{\mathbf{state}[dwt\_depth]}$$

- if level > 0:

$$subband\_width(level, comp) = pw // 2^{\mathbf{state}[dwt\_depth]-level+1}$$

$$subband\_height(level, comp) = ph // 2^{\mathbf{state}[dwt\_depth]-level+1}$$

Note: In the encoding process, these padded dimensions can be achieved by padding the video component data up to the padded dimensions and applying the forward Discrete Wavelet Transform (the inverse of the operations defined in Section 14.3). Any values can be used for the padded data, although the choice will affect wavelet coefficients at the right and bottom edges of the subbands. Good results, in compression terms, are obtained by using edge extension. A poor choice of padding can cause visible artifacts near the bottom and right edges at high levels of compression.

13.2 Inverse Quantization

This section defines the operation of inverse quantization, which scales the magnitude of unpacked wavelet coefficients according to a pre-determined factor. The inverse quantization operation is common to both the low delay and core syntax.

The *inverse_quant*() function shall be as defined as follows:

inverse_quant(quantized_coeff, quant_index):	Ref
magnitude = |quantized_coeff|	
if(magnitude != 0):	
magnitude *= quant_factor(quant_index)	13.2.1
magnitude += quant_offset(quant_index)	13.2.1
magnitude += 2	
magnitude //= 4	
return sign(quantized_coeff) * magnitude	

Notes:

1. VC-2 quantization is an integer approximation of dead-zone quantization, in which a value `x` is quantized as

   `|x|//qf` for `x>=0`, or

   `-(|x|//qf)` for `x<0`.

Since this process involves rounding down, the inverse quantization process adds an offset to non-zero values after multiplying up by `qf`. This produces a reconstruction value closer on average to the original value.

2. The pseudocode description separates inverse quantization from decoding. However, since dead-zone quantization is used, the `inverse_quant()` function must compute the magnitude. Hence it is more efficient to first decode the coefficient magnitude, then inverse quantize, and then decode the coefficient sign.

3. In the low delay syntax, the quantization index is limited to 7 bits; i.e., a maximum value of 127. In the core syntax this value will suffice for 8-bit data and a four level transform, but the maximum value required will depend in general upon the video bit depth, the type of wavelet filter and the transform depth. A maximum value of 127 will account for most practical situations.

13.2.1 Quantization Factors and Offsets

This section defines the operation of the *`quant_factor`*`()` and *`quant_offset`*`()` functions for performing inverse quantization.

Quantization factors shall be determined as follows:

```
quant_factor(index):
   base = 2^(index//4)
   if((index%4) == 0):
      return (4 * base)
   else if((index%4) == 1):
      return(((503829 * base) + 52958) // 105917)
   else if((index%4) == 2):
      return(((665857 * base) + 58854) // 117708)
   else if((index%4) == 3):
      return(((440253 * base) + 32722) // 65444)
```

The quantization offsets shall be determined as follows:

```
quant_offset(index):
   if(index == 0):
      offset = 1
   else if(index == 1):
      offset = 2
   else:
      offset = (quant_factor(index) + 1)//2
   return offset
```

The value of `index` passed to the *quant_offset()* and *quant_factor()* functions shall be greater than or equal to zero.

13.3 DC Subband Prediction

This section defines the operation of the *dc_prediction*(band) process for reconstructing values within DC (0-LL) bands using spatial prediction. This function shall be applied for DC subbands within both the core and the low delay syntax.

This function may be applied once all coefficients within the DC band have been inverse quantized, or applied progressively to each coefficient as soon as it has been inverse quantized to maintain low delay operation, if required.

DC values shall be derived by spatial prediction using the mean of the three values to the left, top-left and above a coefficient (where available).

The DC subband prediction process shall be defined as follows:

```
dc_prediction(band):
   for y = 0 to height(band) - 1:
      for x = 0 to width(band) - 1:
         if(x > 0 and y > 0):
            prediction = mean(band[y][x-1], band[y-1][x-1], band[y-1][x])
         else if (x > 0 and y == 0):
            prediction = band[0][x - 1]
         else if (x == 0 and y > 0):
            prediction = band[y - 1][0]
         else:
            prediction = 0
         band[y][x] += prediction
```

13.4 Core Syntax Wavelet Coefficient Unpacking

This section defines the overall operation of the *core_transform_data(comp)* process for unpacking the set of coefficient subbands corresponding to a video component (Y, C_1 or C_2) in the core VC-2 syntax.

In the VC-2 core syntax, subband data shall be entropy coded. It shall be arranged by level and orientation, from level 0 up to level **state**[dwt_depth].

Notes:

1. Where arithmetic coding is used, the unpacking process for each subband depends upon data from the *parent* subband: i.e. the subband of the same orientation in the next lower level. (The subband of the same orientation in the next higher level being the *child* subband.) Unpacking an individual subband therefore requires prior unpacking of the parent subband, and of its parent, and so on until level 1 is reached (unpacking level 1 subbands does not depend upon the single level 0 DC band).

2. The data for each subband consists of a subband header and a block of coded coefficient data (see Section 12.4.2). The subband header contains a length code giving the number of bytes of the block of coded data. The transform data can therefore be parsed without invoking entropydecoding at all, since the length codes allow a parser to skip from one subband header to the next.

13.4.1 Core Syntax Transform Data

The overall *core_transform_data()* decoding process shall be defined as follows:

***core_transform_data*(comp):**	**Ref**
coeff_data = *initialize_wavelet_data*(comp)	13.1.1
byte_align()	
subband(coeff_data, 0, LL)	13.4.2
for level = 1 **to state**[dwt_depth]:	
for each orient **in** HL, LH, HH:	
byte_align()	
subband(coeff_data, level, orient)	13.4.2
return coeff_data	

This process returns a fully populated array of subband data for a given picture component.

13.4.2 Subbands

This section defines the process for unpacking coefficients within a subband at a specified level and with specified orientation.

The overall process shall consist of reading a byte-aligned header for each subband, including a length code for the subsequent arithmetically coded data. Subband data shall be initialized to zero. If the length code is zero, the subband shall be skipped and all data shall remain set to zero. DC subbands shall be predicted, and so must additionally be reconstructed.

The subband unpacking process shall be defined as follows:

***subband*(coeff_data, level, orient):**	**Type**	**Ref**
length = *read_uint*()	**UInt**	
zero_subband_data(coeff_data[level][orient])		13.4.2.1
if(length == 0):		
byte_align()		
else:		
quant_index = *read_uint*()	**UInt**	
byte_align()		
subband_coeffs(coeff_data,level,orient,length, quant_index)		13.4.2.2
if(level == 0):		
dc_prediction(coeff_data[level][orient])		13.4.2.3

13.4.2.1 Zero Subband

The *zero_subband_data()* process shall set all coefficients in a given subband to zero. It shall be defined as follows:

```
zero_subband_data(band):
    for y = 0 to height(band) – 1:
        for x = 0 to width(band) – 1:
            band[y][x] = 0
```

Note: When a subband or codeblock is skipped, this process ensures that all coefficients within the codeblock remain zero.

13.4.2.2 Non-Skipped Subbands

Data within subbands may be split into one or more rectangular codeblocks (see Section 13.4.3). Codeblocks shall be scanned in raster order within each subband, and coefficients shall be scanned in raster order within each codeblock.

The *subband_coeffs()* process shall be defined as follows:

subband_coeffs(coeff_data, level, orient, length, quant_index):	Ref
state[bits_left] = 8 * length	
if(*using_ac*() == **True**):	10.4.1
ctx_labels = [SIGN_ZERO, SIGN_POS, SIGN_NEG, ZPZN_F1, ZPNN_F1, ZP_F2, ZP_F3, ZP_F4, ZP_F5, ZP_F6+, NPZN_F1, NPNN_F1, NP_F2, NP_F3, NP_F4, NP_F5, NP_F6+, COEFF_DATA, ZERO_BLOCK, Q_OFFSET_FOLLOW, Q_OFFSET_DATA, Q_OFFSET_SIGN]	13.4.4.4
initialize_arithmetic_decoding(ctx_labels)	B.2.2
for cy = 0 to **state**[codeblocks_y][level] - 1:	
for cx = 0 to **state**[codeblocks_x][level] - 1:	
codeblock(coeff_data, level, orient, cx, cy, quant_index)	13.4.3.2
flush_inputb()	A.3.1

Note: The context labels are unique values (Section 5.4.1) used to access context probabilities sets within the arithmetic decoding engine. The precise value that the labels take is implementation defined: in a real implementation they could be unique numbers or memory pointers, but in this standard labels with indicative names are used. The key to the context label names is explained in Section 13.4.4.4.

13.4.3 Subband Codeblocks

This section defines the operation of the process:

codeblock(band, parent, level, orient, cx, cy, quant_index).

This process shall unpack coefficients within the codeblock at position (cx, cy). The dimensions of the codeblock shall be as defined in Section 13.4.3.1. The process for unpacking coefficients within the codeblock given these dimensions shall be as defined in Section 13.4.3.2.

13.4.3.1 Codeblock Dimensions

Each codeblock shall cover coefficients in the horizontal range cb_left to cb_right-1 and in the vertical range cb_top to cb_bottom-1 where these values shall be defined by the functions:

```
cb_left(cx,band,level)  = (width(band)*cx)//state[codeblocks_x][level]
cb_right(cx,band,level) = (width(band)*(cx + 1))//state[codeblocks_x][level]
cb_top(cy,band,level)= (height(band)*cy)//state[codeblocks_y][level]
cb_bottom(cy,band,level)= (height(band)*(cy + 1))//state[codeblocks_y][level]
```

where cx and cy are the codeblock coordinates within a subband.

13.4.3.2 Codeblock Unpacking Loop

The codeblock unpacking process shall be defined by:

***codeblock*(coeff_data, level, orient, cx, cy, quant_index):**	**Ref**
skipped = *zero_flag*(level)	13.4.3.3
if(skipped == **False**):	
quant_index += *codeblock_quant_offset*()	13.4.3.4
for y = *cb_top*(cy,band,level) **to** *cb_bottom*(cy,band,level) - 1:	
for x = *cb_left*(cx,band,level) **to** *cb_right*(cx,band,level) - 1:	
coeff_unpack(coeff_data, level, orient, x, y, quant_index)	13.4.4

If the codeblock is skipped, then coefficients within that codeblock shall remain zero.

The function codeblock_quant_offset() returns a signed value, but the quantizer offset values coded in the stream shall be constrained so that the reconstructed value of quant_index shall be non-negative.

Note: Codeblock quantizers are encoded differentially in the stream, and the value of quant_index is modified by this function (all variables are passed by reference). A decoder ought to check that the reconstructed value of quant_index falls within the bounds that it supports.

13.4.3.3 Skipped Codeblock Flag

The operation of the *zero_flag*() process shall be as follows:

```
zero_flag(level):
   num_blocks = state[codeblocks_x][level] * state[codeblocks_y][level]
   if(num_blocks == 1):
      return False
   else if(using_ac() == True):
      return read_boola(ZERO_BLOCK)
   else :
      return read_boolb()
```

If the number of codeblocks for a subband at a given level is 1, then *zero_flag*() shall return **False**.

If arithmetic coding is employed, then the zero flag shall be decoded using the context probability indicated by the ZERO_BLOCK label, as per Annex A.4.1.

13.4.3.4 Codeblock Quantizer Offset

The *codeblock_quant_offset()* process shall be defined as follows:

```
codeblock_quant_offset():
   if( state[code_block_mode] == 0 ):
      return 0
   else if( using_ac() == True ):
      return read_sinta(quant_context_probs())
   else :
      return read_sintb()
```

where *quant_context_probs*() shall return the context probability label set:

{[Q_OFFSET_FOLLOW], Q_OFFSET_DATA, Q_OFFSET_SIGN}

13.4.4 Subband Coefficients

This section defines the operation of the process.

coeff_unpack(coeff_data, level, orient, x, y, quant_index)

for unpacking an individual coefficient in position (x, y) in the subband coeff_data[level][orient].

Unpacking a coefficient shall make use of entropy decoding (VLC coding or arithmetic coding according to the picture type), inverse quantization and, in the case of DC (level 0) bands, neighborhood prediction.

Arithmetic coding uses a highly compact set of context probabilities, with magnitudes contextualized

depending on whether parent values and neighboring values are zero or non-zero. See Annex B.2 for a definition of the VC-2 arithmetic decoder.

The process for subband coefficient unpacking shall comprise up to four stages:

1. (for arithmetic coding only) determination of the magnitude context, based on whether the parent or neighbouring values are zero,
2. (for arithmetic coding only) determination of the sign context, based on the predicted sign value,
3. entropy decoding of the quantized coefficient value, and
4. inverse quantization of the quantized value.

The *coeff_unpack*() process shall be defined as follows:

***coeff_unpack*(coeff_data, level, orient, x, y, quant_index):**	**Ref**
if (*using_ac()* == **True**):	Table 9.2
parent = *zero_parent*(coeff_data, level, orient, x, y)	13.4.4.1
nhood = *zero_nhood*(coeff_data[level][orient], x, y)	13.4.4.2
sign_pred = *sign_predict*(coeff_data[level][orient], orient, x, y)	13.4.4.3
context_prob_set = *select_coeff_ctxs*(nhood, parent, sign_pred)	13.4.4.4
quant_coeff = *read_sinta*(context_prob_set)	A.4.3.3
else:	
quant_coeff = *read_sintb*()	A.3.3
coeff_data[level][orient][y][x]= *inverse_quant*(quant_coeff,quant_index)	13.2

13.4.4.1 Zero Parent

The function *zero_parent*(coeff_data,level,orient, x, y) shall return a boolean flag indicating whether the parent value of a coefficient in a subband is zero. The parent coefficient is the co-located coefficient in the parent subband if there is one. There shall be deemed to be a parent subband if *level* is greater than 1.

Note: Levels 0 and 1 have the same number of coefficients. Thus the first level that can be used as a parent is level 1, with level 2 as its child.

The parent value shall be determined as follows:

***zero_parent*(coeff_data, level, orient, x, y):**
if(level >= 2):
parent = coeff_data[level - 1][orient][y//2][x//2]
else:
parent = 0
return (parent == 0)

13.4.4.2 Zero Neighborhood

The *zero_nhood*(band, x, y) function shall return a boolean value that defines whether neighboring values are all zero for a given coefficient at location (x, y) in a given subband.

The zero neighborhood value shall be determined as follows:

```
zero_nhood(band, x, y):
   if(y > 0 and x > 0):
      if( (band[y-1][x-1]) != 0 or band[y][x-1] != 0) or
           band[y-1][x] != 0  ):
         return False
   else if(y > 0 and x == 0):
      if(band[y-1][0] != 0):
         return False
   else if(y == 0 and x > 0):
      if(band[0][x-1] != 0):
         return False
   return True
```

Note: The value **True** is returned if the sample on the previous line, the previous sample and the previous sample on the previous line are all zero, with zero also assumed to be the value of non-existent points.

13.4.4.3 Sign Prediction

The *sign_predict*() function shall return a prediction for the sign of the current pixel.

Note: Correlation within subbands depends upon orientation, and so this is taken into account in forming the prediction.

For vertically oriented (HL) bands, the predictor shall be the sign of the coefficient above the current coefficient; for horizontally oriented (LH) bands, the predictor shall be the sign of the coefficient to the left.

The prediction shall be used only for the conditioning of the sign contexts (see Annex A.4.3.1).

The sign prediction value shall be determined as follows:

```
sign_predict(band, orient, x, y):
   if(orient == HL and y == 0):
      return 0
   else if (orient == HL and y > 0):
      return sign(band[y-1][x])
   else if(orient == LH and x == 0):
      return 0
   else if(orient == LH and x > 0):
      return sign(band[y][x-1])
   else:
      return 0
```

13.4.4.4 Coefficient Context Selection

This section defines the coefficient context selection function which returns a map of context probability labels for decoding a coefficient value. Context probabilities shall be as defined in Annex A.4.1.

The map `m` shall comprise three elements, accessed by the labels: `FOLLOW`, `DATA`, and `SIGN`, where:

- `m[FOLLOW]` is an array of labels
- `m[DATA]` is a label
- `m[SIGN]` is a label

The elements of the map returned by *`select_coeff_ctxs`*`(zero_nhood, parent, sign_pred)` shall be as defined in Table 13.1, for values of `zero_nhood`, `parent`, `sign_pred`, where the context labels (e.g., `ZPZN_F1`, `ZP_F2`, `COEFF_DATA`, and `SIGN_ZERO`) correspond to context probabilities (i.e., 16-bit unsigned integers) stored in the decoder state as defined in Annex A.4.1.

The three columns on the left of Table 13.1 represent the three inputs to the coefficient context selection function. The output of the function is a map with three elements. These three elements are accessed by the labels `FOLLOW`, `DATA`, and `SIGN`. The values of the three elements of the map are defined in the rightmost column of the table for each set of inputs.

Note: The follow context probability sets are arrays indexed from zero as per Annex A.4.3. Note also that parent values affect the context of all Follow bits, and that neighbor values only affect the context of the first follow bit. A common data context probability is used for all coefficients.

Key to interpretation of the label names:

ZP – zero parent, NP – non-zero parent,

ZN - zero neighbor, NN – non-zero neighbor

Fn – follow bit, bin N (n+ means bin n and higher)

Table 13.1 – Subband coefficient context probability sets

zero_parent	zero_nhood	sign	context map	
			Element	value
True	True	0	FOLLOW	[ZPZN F1, ZP F2, ZP F3, ZP F4, ZP F5, ZP F6+]
			DATA	COEFF_DATA
			SIGN	SIGN_ZERO
True	True	<0	FOLLOW	[ZPZN F1, ZP F2, ZP F3, ZP F4, ZP F5, ZP F6+]
			DATA	COEFF_DATA
			SIGN	SIGN_NEG
True	True	>0	FOLLOW	[ZPZN_F1, ZP_F2, ZP_F3, ZP F4, ZP F5, ZP F6+]
			DATA	COEFF_DATA
			SIGN	SIGN_POS
True	False	0	FOLLOW	[ZPNN F1, ZP F2, ZP F3, ZP F4, ZP F5, ZP F6+]
			DATA	COEFF_DATA
			SIGN	SIGN_ZERO
True	False	<0	FOLLOW	[ZPNN F1, ZP F2, ZP F3, ZP F4, ZP F5, ZP F6+]
			DATA	COEFF_DATA
			SIGN	SIGN_NEG
True	False	>0	FOLLOW	[ZPNN F1, ZP F2, ZP F3, ZP F4, ZP F5, ZP F6+]
			DATA	COEFF_DATA
			SIGN	SIGN_POS
False	True	0	FOLLOW	[NPZN F1, NP F2, NP F3, NP F4, NP F5, NP F6+]
			DATA	COEFF_DATA
			SIGN	SIGN_ZERO
False	True	<0	FOLLOW	[NPZN F1, NP F2, NP F3, NP F4, NP F5, NP F6+]
			DATA	COEFF_DATA
			SIGN	SIGN_NEG
False	True	>0	FOLLOW	[NPZN F1, NP F2, NP F3, NP F4, NP F5, NP F6+]
			DATA	COEFF_DATA
			SIGN	SIGN_POS
False	False	0	FOLLOW	[NPNN_F1, NP_F2, NP_F3, NP_F4, NP F5, NP F6+]

			DATA	COEFF_DATA
			SIGN	SIGN_ZERO
False	**False**	<0	FOLLOW	[NPNN F1, NP F2, NP F3, NP F4, NP F5, NP F6+]
			DATA	COEFF_DATA
			SIGN	SIGN_NEG
False	**False**	>0	FOLLOW	[NPNN F1, NP F2, NP F3, NP F4, NP F5, NP F6+]
			DATA	COEFF_DATA
			SIGN	SIGN_POS

13.5 VC-2 Low Delay Wavelet Coefficient Unpacking

This section defines the stream syntax that shall be used for low delay profiles. This section defines the syntax and parsing operations only; picture decoding operations are defined in Section 14.

In low delay operation, the VC-2 syntax shall partition the wavelet coefficients into a number of slices, from all subbands, corresponding to localized areas of the picture (see Figure 8.4).

A slice shall meet the following requirements:

1. A single quantizer, weighted for each subband by a quantization matrix, shall be used for quantization of the coefficients in each slice.
2. All wavelet coefficients shall be entropy-coded using variable-length coding alone. Arithmetic coding shall not be used.
3. The number of bytes used per slice shall be the same, to within one byte, for each slice in a picture.
4. Each picture may change the slice parameters within the picture by setting the relevant wavelet transform parameters (Section 12.3.4).

Notes:

1. The slice structure implies that in practice incremental picture decoding can be easily achieved without accumulating an entire picture data set, yielding a decoding delay proportional to the height of the slices. (The actual achievable delay could be more than one slice height due to vertical filtering delay).
2. Using a fixed number of bits per slice does impact on compression efficiency but simplifies both encoder and decoder hardware, and assists a chain of multiple encoders and decoders using the same slice parameters in producing identical coding decisions and hence no cascading loss. These factors are of great significance in a professional environment.

13.5.1 Overall Process

The low delay transform data process shall be defined as follows:

***low_delay_transform_data*():**	**Ref**
state[y_transform] = *initialize_wavelet_data*(Y)	13.1.1
state[c1_transform] = *initialize_wavelet_data*(C1)	13.1.1
state[c2_transform] = *initialize_wavelet_data*(C2)	13.1.1
For sy = 0 **to** **state**[slices_y] – 1:	
for sx = 0 **to** **state**[slices_x] – 1:	
slice(sx, sy)	13.5.2
dc_prediction(**state**[y_transform][0][LL])	13.3
dc_prediction(**state**[c1_transform][0][LL])	13.3
dc_prediction(**state**[c2_transform][0][LL])	13.3

Note: In general, slices cannot be decoded entirely in isolation since DC values at the top and left edges of a slice are predicted from DC values in previously decoded slices, which must therefore be retained.

13.5.2 Slice Unpacking

This section defines the operation of the *slice*(sx, sy) process for parsing the slice with coordinates (sx, sy). Each slice shall contain the relevant coefficients from all subbands and for all components. Luma data shall be unpacked first, and shall be followed by the color difference data, in which the color difference coefficients shall be interleaved. A length code shall allow the luma and color difference coefficients each to be terminated early, with remaining values set to zero by means of bounded read operations.

The overall slice unpacking process shall be defined as follows:

***slice*(sx, sy):**	**Type**	**Ref**
slice_bits_left = 8**slice_bytes*(sx, sy)		13.5.3
qindex = *read_nbits(7)*	UInt	
slice_bits_left -= 7		
slice_quantizers(qindex)		13.5.4
length_bits = intlog$_2$(8**slice_bytes*(sx, sy)-7)		
slice_y_length = *read_nbits*(length_bits)		
slice_bits_left -= length_bits		
state[bits_left] = slice_y_length		
luma_slice_band(0, LL, sx, sy)		13.5.5.2
for level = 1 to **state**[dwt_depth]:		
for each orient **in** HL, LH, HH:		

luma_slice_band(level, orient, sx, sy)		13.5.5.2
flush_inputb()		A.3.1
slice_bits_left -= slice_y_length		
state[bits_left] = slice_bits_left		
color_diff_slice_band(0, LL, sx, sy)		13.5.5.3
for level = 1 to **state**[dwt_depth]:		
for each orient **in** HL, LH, HH:		
color_diff_slice_band(level,orient,sx,sy)		13.5.5.3
flush_inputb()		A.3.1

slice_y_length shall satisfy the following condition:

slice_y_length • 8**slice_bytes*(sx, sy)-7-length_bits

Note: Slice decoding makes use of bounded read functions, which return 1 when **state**[bits_left] is zero. This means that remaining coefficients are set to 0, since a solitary 1 is the VLC for 0. The logic of slice decoding applies this twice in each slice: once for luma coefficients, initializing the number of bits left to slice_y_length, and a second time for the color difference coefficients, initializing to the number of bits remaining.

13.5.3 Determining the Number of Bytes in a Slice

The *slice_bytes*() function shall be defined as follows:

```
slice_bytes(sx, sy):
    slice_number = sy*state[slices_x] + sx
    bytes = ((slice_number+1)*state[slice_bytes_numerator])//
        state[slice_bytes_denominator]
    bytes -= (slice_number*state[slice_bytes_numerator])//
        state[slice_bytes_denominator]
    return bytes
```

Note: This function produces an integer value which will on average be the ratio of

state[slice_bytes_numerator] to **state**[slice_bytes_denominator].

In many applications, this ratio will not be an integer number of bytes, and the number of bytes per slice will vary by 1 byte from time to time. This allows the low delay syntax to support any rational compression ratio exactly, so that a picture can be compressed to an arbitrary number of bytes.

13.5.4 Setting Slice Quantizers

This section defines how quantizers for individual subbands shall be determined from the quantization matrix and the quantization index. The *slice_quantizers*() process shall be defined as follows:

```
slice_quantizers(qindex):
    state[quantizer][0][LL] = max(qindex – state[quant_matrix][0][LL], 0)
    for level = 1 to state[dwt_depth]:
        for each orient in HL, LH, HH:
            qval = max(qindex – state[quant_matrix][level][orient], 0)
            state[quantizer][level][orient] = qval
```

Note: The non-negative quantization matrix values are subtracted from the slice quantizer value, and so a higher value in the quantization matrix represents a lower quantization index and thus a lower degree of quantization. The quantization index value is also clipped to 0 so that it is non-negative. This ensures that as many subbands as possible within the slice can be coded losslessly.

13.5.5 Slice Subbands

This section defines the operation of the *luma_slice_band*(level, orient, sx, sy) for unpacking individual luma slice subbands, and *color_diff_slice_band*(level, orient, sx, sy) for unpacking individual color difference slice subbands.

13.5.5.1 Slice Subband Area

The rectangular set of coefficients covered by a slice component (Y, C_1 and C_2) with index (sx,sy) is demarcated by the values *slice_left, slice_right, slice_top, slice_bottom*, defined as fractions of the subband dimensions by the following functions:

```
slice_left(sx,c,level)   = (subband_width(level,c) * sx)//state[slices_x]
slice_right(sx,c,level)  = (subband_width(level,c) * (sx +
                           1))//state[slices_x]
slice_top(sy,c,level)    = (subband_height(level,c) * sy)//state[slices_y]
slice_bottom(sy,c,level) = (subband_height(level,c) * (sy +
                           1))//state[slices_y]
```

where *c* is Y for luma coefficients, and C_1 or C_2 for color difference coefficients.

Notes:

1. The slice subband area formulae correspond to the codeblock area formulae for the core syntax (Section 13.4.3.1).
2. Slice subbands can change dimension by 1 from one slice to another if **state**[slices_x] or **state**[slices_y] do not divide the horizontal or vertical dimensions exactly.
3. Color difference slice subbands might not have exactly the scaled dimensions of the luma slice subband, since the **state**[slices_x] and **state**[slices_y] values could exactly divide luma dimensions but not color difference dimensions; and color difference components might receive different padding, depending on the transform depth.
4. These issues can easily be avoided in particular applications by choosing suitable values for the transform depth and **state**[slices_x] and **state**[slices_y].

13.5.5.2 Luma Slice Subband Data

The process for unpacking luma slice coefficients shall be defined as follows:

luma_slice_band(level, orient, sx, sy):	Ref
for y = *slice_top*(sy,Y,level) **to** *slice_bottom*(sy,Y,level) – 1:	
for x = *slice_left*(sx,Y,level) **to** *slice_right*(sx,Y,level) – 1:	
val = *read_sintb()*	A.3.3
qi = **state**[quantizer][level][orient]	
state[y_transform][level][orient][y][x] = *inverse_quant*(val, qi)	14.2

13.5.5.3 Color difference Slice Subband Data

Color difference slice subband coefficients shall follow luma coefficients within each slice. The two color difference components shall be interleaved coefficient by coefficient. The process for unpacking color difference slice coefficients shall be defined as follows:

color_diff_slice_band(level, orient, sx, sy):	Ref
qi = **state**[quantizer][*level*][*orient*]	
for y = *slice_top*(sy,C1,level) **to** *slice_bottom*(sy,C1,level) – 1:	
for x = *slice_left*(sx,C1,level) **to** *slice_right*(sx,C1,level) – 1:	
qi = **state**[quantizer][level][orient]	
val = *read_sintb()*	A.3.3
state[c1_transform][level][orient][y][x] = *inverse_quant*(val, qi)	14.2
val = *read_sintb()*	A.3.3
state[c2_transform][level][orient][y][x] = *inverse_quant*(val, qi)	14.2

14 Picture Decoding

This section defines the processes for decoding a picture from a VC-2 stream. Picture decoding depends upon correctly parsing the VC-2 stream, and decoding operations are dependent on decoding the VC-2 stream and picture syntax (Sections 11 and 12) and unpacking the coefficient data (Section 13).

14.1 Overall Picture Decoding Process

This section defines the processes for decoding a picture from a VC-2 stream.

During the decoding process picture data from the current picture being decoded shall be stored in the **state**[current_picture] state variable. This is a map with an index *pic_num* and data arrays

state[current_picture][Y],
state[current_picture][C1], and
state[current_picture][C2]

representing the video components of the picture.

After decoding the decoded picture is returned to the decoding application.

The *picture_decode*() process shall be defined as follows:

***picture_decode*():**	**Ref**
state[current_picture] = {}	
state[current_picture][pic_num] = **state**[picture_number]	12.1
inverse_wavelet_transform()	14.2
clip_picture(**state**[current_picture])	14.4
offset_picture(**state**[current_picture])	14.4
return state[current_picture]	

14.2 Picture IDWT

The inverse discrete wavelet transform process shall consist of transforming the wavelet coefficients for each of the video components. It shall be defined as follows:

***inverse_wavelet_transform*():**	**Ref**
state[current_picture][Y] = *idwt*(**state**[y_transform])	14.3
state[current_picture][C1] = *idwt*(**state**[c1_transform])	14.3
state[current_picture][C2] = *idwt*(**state**[c2_transform])	14.3
for c **in** Y, C1, C2 :	
idwt_pad_removal(**state**[current_picture][c], c)	14.3.4

14.3 Component IDWT

This section defines the process *idwt*(coeff_data) for reconstructing picture component data from decoded subband data *coeff_data* using the inverse discrete wavelet transform (IDWT). The IDWT shall be invoked in the picture decoding process only after successful unpacking of the wavelet coefficient data as defined in Section 13.

The IDWT process shall return a pixel array corresponding to a single reconstructed video component.

Since wavelet filtering operates on both rows and columns of two-dimensional arrays independently, it is useful to define operators row(a,k) and column(a,k) for extracting rows and columns with index k from a 2-dimensional array a:

If b = row(a,k), then b[r] is a reference to the value of a[k][r].

This means that modifying the value of b[r] for any index r modifies the value of a[k][r].

If b = column(a,k), then b[r] is a reference to the value of a[r][k].

This means that modifying the value of b[r] for any index r modifies the value of a[r][k].

The *idwt* shall be an iterative procedure operating on four subbands (LL, HL, LH and HH) at each iteration stage to produce a new subband (LL). The procedure shall be as follows:

idwt(coeff_data):	Ref
LL_band = coeff_data[0][LL]	
for n = 1 **to** **state**[dwt_depth]:	
new_LL_band = *vh_synthesis*(LL_band, coeff_data[n][HL], coeff_data[n][LH],coeff_data[n][HH])	14.3.1
LL_band = new_LL_band	
return LL_band	

Note: At each stage, the dimensions of the input LL_band will be the same as those of the other input bands, whereas the output dimensions are double those of the input bands.

14.3.1 Vertical and Horizontal Synthesis

This section defines the operation of the vertical and horizontal synthesis process:

vh_synthesis(LL_data, HL_data, LH_data, HH_data).

vh_synthesis() shall return an array of twice the dimensions of each of the input argument arrays.

vh_synthesis() is repeatedly invoked by the IDWT synthesis operation (section 14.3), and operates on each set of four subband data arrays of identical dimensions to produce a new array synth, which shall be returned as the result of the process.

Step 1. synth is a temporary two-dimensional array that shall be initialized so that:

width(synth) = 2 * width(LL_data)

height(synth) = 2 * height(LL_data)

Step 2. The data from the four arrays shall be interleaved in the synth array as follows:

```
…
for y = 0 to (height(synth)//2) - 1:
    for x = 0 to (width(synth)//2) - 1:
        synth[2y][2x] = LL_data[y][x]
        synth[2y][2x + 1] = HL_data[y][x]
        synth[2y + 1][2x] = LH_data[y][x]
        synth[2y + 1][2x + 1] = HH_data[y][x]
…
```

Note: This interleaving enables in-place calculation during the inverse filter process.

Step 3. Data shall be synthesized vertically by operating on each column of data using a one-dimensional filter, and then horizontally by operating on each row of data using a set of one-dimensional filters. The one-dimensional lifting filters used shall be determined by the value of **state**[wavelet_index] according to Tables 14.1 thru 14.7. The process shall be as follows:

…	**Ref**
for x = 0 **to** width(synth) - 1:	
1d_synthesis(column(synth, x))	14.3.2
for y = 0 **to** height(synth) - 1:	
1d_synthesis(row(synth, y))	14.3.2
…	

Step 4. Finally, the synthesized subband data shall implement a bit-shift to remove any accuracy bits. The bit-shift value *filter_bit_shift()* used shall be determined by the value of **state**[wavelet_index] according to Tables 14.1 thru 14.7. The process shall be as follows:

```
…
shift = filter_bit_shift()
if (shift > 0):
    for y = 0 to height(synth) – 1:
        for x = 0 to width(synth) - 1:
            synth[y][x] = (synth[y][x] + (1 << (shift – 1))) >> shift
```

Note: Accuracy bits are added in the encoder by shifting up all coefficients in the LL band prior to applying any filtering (this includes an initial shift of all values in the component data). Adding a small shift before each decomposition stage is the most efficient way of providing additional resolution mitigating aliasing through non-linear rounding effects.

14.3.2 One-Dimensional Synthesis

This section defines the one-dimensional synthesis process. *1d_synthesis*() shall apply to a one-dimensional array of coefficients of even length, consisting of either a row or a column of a two-dimensional integral data array.

The one-dimensional synthesis process shall comprise the application of a number of reversible integer lifting filter operations.

Lifting filtering operations shall be one of four types: Type 1, Type 2, Type 2, and Type 4. Each type of integral lifting filter shall be characterized by four elements:

1. a filter length value L
2. a filter offset value D
3. an array of taps of length L : taps[0], taps[1], … taps[L-1]
4. a scale factor S

The four types of lifting operations shall be defined by the functions:

lift1(A, L, D, taps, S),
lift2(A, L, D, taps, S),
lift3(A, L, D, taps, S), and
lift4(A, L, D, taps, S)

respectively, and shall act upon the values in a one-dimensional array A.

The Type 1 lifting process *lift1***(A, L, D, taps, S)** shall be defined as follows:

```
lift1(A, L, D, taps, S):
  for n = 0 to (length(A)//2) - 1:
    sum = 0
    for i = D to L + D - 1:
      pos = 2*(n + i) – 1
      pos = min(pos, length(A) – 1)
      pos = max(pos, 1)
      sum += taps[i-D]* A[pos]
    if S>0 :
      sum += (1<<(S - 1))
    A[2*n] += (sum >> S)
```

The Type 2 lifting process *lift2***(A, L, D, taps, S)** shall be defined as follows:

```
lift2(A, L, D, taps, S):
  for n = 0 to (length(A)//2) - 1:
    sum = 0
    for i = D to L + D - 1:
      pos = 2*(n + i) – 1
      pos = min(pos, length(A) – 1)
      pos = max(pos, 1)
      sum += taps[i-D] * A[pos]
    if S>0 :
      sum += (1<<(S - 1))
    A[2*n] -= (sum >> S)
```

The Type 3 lifting process *lift3***(A, L, D, taps, S)** shall be defined as follows:

```
lift3(A, L, D, taps, S):
   for n = 0 to (length(A)//2) - 1:
      sum = 0
      for i = D to L + D - 1:
         pos = 2*(n + i)
         pos = min(pos, length(A) – 2)
         pos = max(pos, 0)
         sum += taps[i-D] * A[pos]
      if S>0 :
         sum += (1<<(S - 1))
      A[2*n + 1] += (sum >> S)
```

The Type 4 lifting process *lift4*`(A, L, D, taps, S)` shall be defined as follows:

```
lift4(A, L, D, taps, S):
   for n = 0 to (length(A)//2) - 1:
      sum = 0
      for i = D to L + D - 1:
         pos = 2*(n + i)
         pos = min(pos, length(A) – 2)
         pos = max(pos, 0)
         sum += t[i-D] * A[pos]
      if S>0 :
         sum += (1<<(S - 1))
      A[2*n + 1] -= (sum >> S)
```

1d_synthesis shall apply the sequence of lifting filters specified in Section 14.3.3 corresponding to the value of `state`[wavelet_index], and shall invoke the corresponding lifting processes with the parameters defined.

14.3.2.1 Mathematical Formulation of Lifting Processes (Informative)

The lifting processes defined in the previous section are extremely similar, and careful attention must be paid to the detail of their operation in any implementation. The four different variants arise from two factors: the 'phase' (odd or even) of the lifting operation, and their implementation using integer-only operations, which introduces rounding errors and makes addition and subtraction subtly different.

A lifting operation either modifies the odd coefficients by a linear combination of the even coefficients, or vice-versa. Mathematically, the four types of filter can be described as follows.

Type 1 and Type 2 filtering operations modify the even coefficients by the odd coefficients:

$$A[2n] \mathrel{+}= \left(\sum_{i=-N}^{M} t_i A[2(n+i)-1] + (1 << (s-1)) \right) >> s \qquad \text{(Type 1)}$$

$$A[2n] \mathrel{-}= \left(\sum_{i=-N}^{M} t_i A[2(n+i)-1] + (1 << (s-1)) \right) >> s \qquad \text{(Type 2)}$$

Type 3 and Type 4 lifting filtering operation modify the odd coefficients by the even coefficients:

$$A[2n+1] \mathrel{+}= \left(\sum_{i=-N}^{M} t_i A[2(n+i)] + (1 << (s-1)) \right) >> s \qquad \text{(Type 3)}$$

$$A[2n+1] \mathrel{-}= \left(\sum_{i=-N}^{M} t_i A[2(n+i)] + (1 << (s-1)) \right) >> s \qquad \text{(Type 4)}$$

The distinctions between Type 1 and Type 2 and between Type 3 and Type 4 are necessary because integer division (bit-shifting) is used, rendering the filters non-linear: a Type 1 or Type 3 filter with taps: t not being equivalent to a Type 2 or Type 4 filter with taps: $-t$.

Edge extension is used where the filter would otherwise extend beyond the boundaries of the array. This is slightly different between Types 1 and 2 on the one hand and Types 3 and 4 on the other. This is because even values and odd values must be extended separately to maintain the correct phase (and hence invertibility of the filter). For example, a Type 1 filter must use the values 1 and length(A)-1 at the edges because (assuming the length to be even) these are the odd values nearest the edges.

Further information on wavelet filtering and lifting is provided in Annex G.

14.3.3 Lifting Filter Parameters

The lifting filters and filter bit-shift operations that apply for each value of **`state`**`[wavelet_index]` shall be as specified in Tables 14.1 thru 14.7.

Table 14.1 – `state[wavelet_index] == 0`: Deslauriers-Dubuc (9,7) lifting stages and shift values

Lifting steps	*filter_bit_shift*()
Two lifting stages: 1. Type 2, S = 2, L=2, D=0, taps = [1,1] i.e.: A[2n] -= (A[2n-1] + A[2n+1] + 2) >> 2 2. Type 3, S = 4, L=4, D=-1, taps=[-1,9,9,-1] i.e.: A[2n+1] += (-A[2n-2] + 9A[2n] + 9A[2n+2] - A[2n+4] +8) >> 4	1

Table 14.2 – `state[wavelet_index] == 1`: LeGall (5,3) lifting stages and shift values

Lifting steps	*filter_bit_shift()*
Two lifting stages: 1. Type 2, S = 2, L = 2, D = 0, taps = [1,1] i.e.: A[2n] -= (A[2n - 1] + A[2n + 1] + 2) >> 2 2. Type 3, S = 1, L = 2, D = 0, taps=[1,1] i.e.: A[2n+ 1] += (A[2n] + A[2n + 2] + 1) >> 1	1

Table 14.3 – state[wavelet_index] == 2: Deslauriers-Dubuc (13,7) lifting stages and shift values

Lifting steps	*filter_bit_shift()*
Two lifting stages: 1. Type 2, S = 5, L = 4, D = -1, taps = [-1,9,9,-1] i.e.: A[2n] -= (-A[2n-3] + 9A[2n-1] + 9A[2n+1] - A[2n+3] + 16) >> 5 2. Type 3, S = 4, L = 4, D = -1, taps = [-1,9,9,-1] i.e. A[2n+1] += (-A[2n-2] + 9A[2n] + 9A[2n+2] - A[2n+4] + 8) >> 4	1

Table 14.4 – state[wavelet_index] == 3: Haar filter with no shift

Lifting steps	*filter_bit_shift()*
Two lifting stages: 1. Type 2, S = 1, L = 1, D = 1, taps = [1] i.e. A[2n] -= (A[2n+1] + 1)>>1 2. Type 3, S = 0, L = 1, D = 0, taps = [1] i.e. A[2n+1] += A[2n]	0

Table 14.5 – `state[wavelet_index] == 4`: Haar filter with single shift

Lifting steps	*filter_bit_shift()*
Two lifting stages: 1. Type 2, S = 1, L = 1, D = 1, taps = [1] i.e. A[2n] -= (A[2n+1] + 1)>>1 2. Type 3, S = 0, L = 1, D = 0, taps = [1] i.e. A[2n+1] +=A[2n]	1

Table 14.6 – `state[wavelet_index] == 5`: Fidelity filter

Lifting steps	*filter bit shift*( )
Two lifting stages: 1. Type 3, S = 8, L = 8, D = -3, taps = [-2,10,-25,81,81,-25,10,-2] i.e. A[2n+1] += (-2(A[2n-6] + A[2n+8]) + 10(A[2n-4] + A[2n+6]) - 25(A[2n-2] + A[2n+4]) + 81(A[2n] + A[2n+ 2]) + 128) >> 8 2. Type 2, S = 8, L = 8, D=-3, taps = [-8,21,-46,161,161,-46,21,-8] A[2n] -= (-8(A[2n-7] + A[2n+7]) + 21(A[2n-5] + A[2n+5]) - 46(A[2n-3] + A[2n+3]) + 161(A[2n-1] + A[2n+1]) + 128) >> 8	0

Note: The Fidelity filter was specifically developed by the VC-2 creators using an extensive search process to select a filter set with simple integer filter coefficients limited to 8-bit resolution and offering good alias rejection in both analysis and synthesis operations. The resulting filter is almost mirror symmetric and the good anti-aliasing characteristics provide improved picture quality of the sub-sampled images. The FIR filter coefficient equivalents of the filters are as follows:

The equivalent analysis low-pass FIR filter is:

(-8, 0, 21, 0, -46, 0, 161, 256, 161, 0, -46, 0, 21, 0, -8)/256

and the equivalent analysis high-pass FIR filter is:

(16, 0, -122, 0, 502, 0, -1955, -512, 3491, 2560, -4148, -6400, 4183, 20736, -36702, 20736, 4183, -6400,

-4148, 2560, 3491, -512, -1955, 0, 502, 0, -122, 0, 16)/65536

The synthesis filters swap over the analysis filters above and alternate the signs, thus the equivalent synthesis low-pass FIR filter is:

(-16, 0, 122, 0, -502, 0, 1955, -512, -3491, 2560, 4148, -6400, -4183, 20736, 36702, 20736, -4183, -6400, 4148, 2560, -3491, -512, 1955, 0, -502, 0, 122, 0, -16)/65536

and the equivalent synthesis high-pass FIR filter is:

(8, 0, -21, 0, 46, 0, -161, 256, -161, 0, 46, 0, -21, 0, 8)/256

Table 14.7 – `state[wavelet_index] == 6`: Integer lifting approximation to Daubechies (9,7)

Lifting steps	*filter_bit_shift*()
Four lifting stages: 1. Type 2, S = 12, L = 2, D = 0, taps = [1817,1817] i.e. A[2n] -= (1817A[2n-1] + 1817A[2n+1] + 2048)>>12 2. Type 4, S = 12, L = 2, D = 0, taps = [3616,3616] i.e. A[2n+1] -= (3616A[2n] + 3616A[2n+2] + 2048)>>12 3. Type 1, S = 12, L = 2, D = 0, taps = [217,217] i.e. A[2n] += (217A[2n-1] + 217A[2n+1] + 2048)>>12 4. Type 3, S = 12, L = 2, D = 0, taps = [6497,6497] i.e. A[2n+1] += (6497A[2n] + 6497A[2n+2] + 2048)>>12	1

14.3.4 Removal of IDWT Pad Values

This section defines the decoding process *idwt_pad_removal*(pic, c) for removing extraneous values after performing the IDWT.

Section 13.1.2 requires that subband coefficient data arrays are padded to ensure that the reconstructed data array *pic* dimensions are multiples of $2^{state[dwt\_depth]}$.

Values width and height are defined to be the appropriate dimensions of the component data:

```
if(c == Y):                          width = state[luma_width]
                                     height = state[luma_height]
else if((c == C1) or (c = C2)):      width = state[color_diff_width]
                                     height = state[color_diff_height]
```

All component data pic[j][i] with: i >= width, or j >= height shall be discarded and the reconstructed picture shall be resized to have the defined width and height.

14.4 Picture Output Ranges

Picture data shall be clipped to the upper and lower signal extremes prior to being output. The *clip_picture()* process shall be as defined by the following table:

```
clip_picture(current_picture):
    for each c in Y, C1, C2:
        clip_component(current_picture[c], c)
```

The *clip_component()* process shall be as defined by the following table:

```
clip_component(comp_data, c):
    for y = 0 to height(comp_data)-1:
        for x = 0 to width(comp_data)-1:
            if (c==Y):
                clip(comp_data[y][x], -2^(state[luma_depth]-1), 2^(state[luma_depth]-1)-1)
            else:
                clip(comp_data[y][x], -2^(state[color_diff_depth]-1), 2^(state[color_diff_depth]-1)-1)
```

Picture data shall be offset so as to be output as unsigned integer values. The *offset_picture()* process shall be as defined by the following table:

```
offset_picture(current_picture):
    for each c in Y, C1, C2:
        offset_component(current_picture[c], c)
```

The *offset_component()* process shall be as defined by the following table:

```
offset_component(comp_data, c):
    for y = 0 to height(comp_data)-1:
        for x = 0 to width(comp_data)-1:
            if (c==Y):
                comp_data[y][x] += 2^state[luma_depth]-1
            else:
                comp_data[y][x] += 2^state[color_diff_depth]-1
```

Note: Where the picture data is used on video interfaces such as a SMPTE Serial Digital Interface (SDI), there can be further constraints on the minimum and maximum values.

Annex A VC-2 Data Coding Definitions (Normative)

Data shall be represented in the VC-2 stream using one of four basic methods:

1. fixed-length bit-wise codings,
2. fixed-length byte-wise codings,
3. variable-length codes and
4. arithmetic coding.

This annex defines how data shall be represented in the VC-2 stream and how sequences of bits shall be extracted as values of various types using the aforementioned fundamental data coding types. The extraction of arithmetically encoded data shall require the use of the arithmetic decoding engine defined in Annex B.2.

A.1 Bit Packing and Data Input

This section defines the operation of the *read_bit()*, *read_byte()* and *byte_align()* functions used for direct access to fixed length encodings in the VC-2 stream.

The stream data shall be accessed byte by byte, and a decoder is deemed to maintain a copy of the current byte, **state**[current_byte], and an index to the next bit (in the byte) to be read, **state**[next_bit].

state[next_bit] shall an integer from bit 0 (the least-significant bit) to bit 7 (the most-significant bit). Bits within bytes shall be accessed from the msb first to the lsb last.

A.1.1 Reading a Byte

The *read_byte()* function shall perform the following steps:

1. Set **state**[next_bit] = 7
2. Set **state**[current_byte] to the next unread byte in the stream

A.1.2 Reading a Bit

The *read_bit()* function shall be defined as follows:

```
read_bit():
    bit = (state[current_byte] >> state[next_bit])&1
    state[next_bit] -= 1
    if(state[next_bit] < 0):
        state[next_bit] = 7
        read_byte()
    return bit
```

A.1.3 Byte Alignment

The *byte_align*() function shall be used to discard data in the current byte and begin data access at the next byte, except where the input is already at the beginning of a byte. The *byte_align*() function shall be as defined in the following table:

```
byte_align():
    if(state[next_bit] != 7):
        read_byte()
```

This function shall be used to ensure that a whole number of bytes are read before beginning reading a new stream element: for example, each parse info header within the stream is byte aligned.

A.2 Fixed Length Data

VC-2 defines three fixed length bitstream data encodings as follows:

A.2.1 Boolean

The *read_bool()* function shall return **True** if 1 is read from the stream and **False** otherwise. The *read_bool()* function shall be as defined in the following table:

```
read_bool():
    if(read_bit() == 1):
        return True
    else:
        return False
```

A.2.2 n-bit Unsigned Integer Literal

An n-bit unsigned integer in literal format shall be decoded by extracting n bits in order, using the *read_bit()* function (Annex A.1.2) and placing the first bit in the leftmost position, the second bit in the next position and so on. The resulting value shall be interpreted as an unsigned integer.

The *read_nbits()* function shall be defined as follows:

```
read_nbits(n):
    val = 0
    for i = 0 to n- 1:
        val <<= 1
        val += read_bit()
    return val
```

A.2.3 n-byte Unsigned Integer Literal

A single byte shall be be interpreted as an unsigned integer value in the range 0 to 255. The process defined below reads one or more bytes as needed to create an n-byte value with the most significant byte read first. The *read_uint_lit()* function shall be defined as follows:

```
read_uint_lit(n):
    byte_align()
    return read_nbits(8*n)
```

A.3 Variable-length Codes

Variable-length codes (VLC) shall be used in three ways in the VC-2 data stream.

1. The first use shall be for the representation of header values into the stream.
2. The second use shall be for the representation of wavelet coefficients where arithmetic coding is not used.
3. The third use shall be for binarization in conjunction with arithmetic coding, whereby integer values may be decoded using a binary arithmetic decoder. This is described in Annex A.4.

When used for representing wavelet coefficients, VLCs shall be employed within a data block of known length. It is possible to gain additional compression by early termination; maintaining a count of remaining bits, and returning default values when this length is exceeded. This shall be achieved by the use of the *read_bitb()*, *read_uintb()* and *read_sintb()* functions for reading values from data blocks.

Note: A similar early termination facility is used for arithmetic decoding.

A.3.1 Data Input for Bounded Block Operation

This section defines the operation of the *read_bitb()* and *read_boolb()* processes for reading bits and boolean values from a block of known size, and the *flush_inputb()* process for discarding the remainder of a block of data.

These processes shall use **state**[bits_left] to determine the remaining bits to the end of the block.

The *read_bitb()* function shall be defined as follows:

```
read_bitb():
    if(state[bits_left] == 0):
        return 1
    else:
        state[bits_left] -= 1
        return read_bit()
```

When all bits in the block have been read, then *read_bitb()* shall return 1 by default.

The *read_boolb()* function operates analogously to the *read_bool()* function and shall be defined as follows:

```
read_boolb():
  if(read_bitb() == 1):
     return True
  else:
     return False
```

It is possible that not all data in a block is exhausted after a sequence of read operations. At the end of a sequence of bounded block read operations, the decoder shall flush the block.

The *flush_inputb()* process shall be defined as follows:

```
flush_inputb():
   while(state[bits_left] > 0):
      read_bit()
      state[bits_left] -= 1
```

A.3.2 Unsigned Interleaved Exp-Golomb Codes

This section defines the unsigned interleaved exp-Golomb data format and the operation of the *read_uint()* and the *read_uintb()* functions.

Unsigned interleaved exp-Golomb data shall be decoded to produce unsigned integer values. The format shall consist of two interleaved parts, and each code shall be an odd number of 2K + 1 bits in length.

The K + 1 bits in the even positions (counting from zero) shall be the "follow" bits, and the K bits in the odd positions shall be the "data" bits b_i that are used to construct the decoded value itself. A follow bit value of 0 shall indicate a subsequent data bit, whereas a follow bit value of 1 shall terminate the code, a typical sequence being:

$$0 , x_{K-1} , 0 , x_{K-2} ,\ldots, 0 , x_0 , 1$$

The data bits x_i shall be the binary representation of the first K bits of the (K + 1) bit number (N + 1), where N is the number to be decoded, i.e.

$$N + 1 = b1x_{K-1}x_{K-2}...x_0 = 2^K + x_{K-1}*2^{K-1} + x_{K-2}*2^{K-2} + \ldots + x_0$$

A list of encodings of the first 10 values is shown in Table A.1.

Table A.1 – Example conversions from unsigned interleaved exp-Golomb-coded values to unsigned integers

Bit sequence	Decoded value
1	0
001	1
011	2
00001	3
00011	4
01001	5
01011	6
0000001	7
0000011	8
0001001	9

Although apparently complex, the interleaving ensures that the code has a very simple decoding loop. The *read_uint()* function shall return an unsigned integer value and shall be defined as follows:

```
read_uint():
   value = 1
   while(read_bit() == 0):
      value <<= 1
      if(read_bit() == 1):
         value += 1
   value -= 1
   return value
```

Note: Conventional exp-Golomb coding places all follow bits at the beginning as a prefix. This is easier to read, but requires that a count of the prefix length be maintained. Values can only be decoded in two loops – the prefix followed by the data bits. Interleaved exp-Golomb coding allows values to be decoded in a single loop, without the need for a length count.

The *read_uintb()* function is identical to *read_uint()* function except that the block-bounded read operation is employed and shall be defined as follows:

```
read_uintb():
    value = 1
    while(read_bitb() == 0):
        value <<= 1
        if(read_bitb() == True):
            value += 1
    value -= 1
    return value
```

Note: When **state**[bits_left] == 0, all subsequent values read by *read_uintb()* will be zero.

A.3.3 Signed Interleaved Exp-Golomb Codes

This section defines the signed interleaved exp-Golomb data format and the operation of the *read_sint()* and *read_sintb()* functions.

The code for the signed interleaved exp-Golomb data format shall consist of the unsigned interleaved exp-Golomb code for the magnitude, followed by a sign bit for non-zero values as shown in Table A.2.

Table A.2 – Example conversions from signed interleaved exp-Golomb-coded values to signed integers

Bit sequence	Decoded value
000111	-4
000011	-3
0111	-2
0011	-1
1	0
0010	1
0110	2
000010	3
000110	4

The decoding process for *read_sint()* shall be as follows.

```
read_sint():
    value = read_uint()
    if(value != 0):
        if(read_bit() == 1):
            value = -value
    return value
```

The *read_sintb()* function shall be identical to *read_sint()* except that the block-bounded read operation shall be employed and shall be as defined as follows:

```
read_sintb():
    value = read_uintb()
    if(value != 0):
        if(read_bitb() == 1):
            value = -value
    return value
```

Note: When **state**[bits_left] == 0, all subsequent values read by *read_sintb()* will be zero.

A.4 Parsing of Arithmetic Coded Data

This section defines the operations for reading and writing arithmetic coded data. These operations shall make use of the arithmetic decoding engine defined in annex B.2. The operations assume that the value of the state variable **state**[bits_left] has been set so that bounded read operations are used during the renormalization process (Annex B.2.5).

Arithmetically coded data is present in the VC-2 stream in data blocks which shall consist of a whole number of bytes and shall be byte aligned. Where arithmetic coding is used, each such block shall be preceded by data which includes a length code, which shall define the length of the data block in bytes.

The function *initialize_arithmetic_decoding()* (Annex B.2.2) shall then initialize the arithmetic decoder. Once the arithmetic decoder has been initialized, Boolean and Integer values may be extracted.

After all values in a particular arithmetic coded block have been parsed, the remaining data shall be purged by implementing *flush_inputb()* (Annex A.3.1).

A.4.1 Context Probabilities

Binary values shall be extracted by using one of a set of probabilities. Which probability is selected depends on the context for that symbol. A context classifies symbols into different types based on data decoded so far: for example, symbols occurring in coefficients whose neighbours are all zero could constitute one context.

A context probability shall be a 16 bit unsigned integer value representing the probability of a bit being 0 in a given context, where zero probability is represented by 0x0, and equal likelihood by 0x8000. The process for initializing and updating context probabilities shall be as defined in Annex B.2.1.

Note: Probability 1, or certainty, would be represented by the 17-bit number 0x10000. This value, and probability 0 = =0x0, can never be attained due to the operation of the probability update process (Annex B.2.6).

Different context probabilities shall be employed for extracting binary values, based on the values of previously decoded data. Each context probability shall be updated by the arithmetic decoding engine to track statistics after it has been used to extract a value.

The set of contexts probabilities shall be defined by **`state`**`[context_probs]`, and an individual context shall be accessed via a keyword label `l` i.e. **`state`**`[context_probs]` is a map and the context value shall be **`state`**`[context_probs][l]` for a label `l`.

The array of context probability labels to be used in arithmetic decoding shall be passed to the arithmetic decoding engine at initialization (Annex B.2.2).

A.4.2 Arithmetic Decoding of Boolean Values

Given a context probability label `l`, the arithmetic decoding engine shall support a function *`read_boola(`*`l`*`)`*, defined in Annex B.2.4, which shall return the boolean value: **`True`** or **`False`**.

A.4.3 Arithmetic Decoding of Integer Values

This section defines the operation of the *`read_sinta(`*`context_prob_set`*`)`* and *`read_uinta(`*`context_prob_set`*`)`* functions for extracting integer values from a block of arithmetically coded data.

A.4.3.1 Binary Coding and Contexts

When arithmetic coding is used, signed and unsigned integer values shall be represented as a string of binary values using interleaved exp-Golomb coding as per annexes A.3.2 and A.3.3, each of which is extracted from the stream using binary arithmetic coding with a context assigned to each bit. As a result the *`read_sinta()`* and *`read_uinta()`* processes shall be identical to the *`read_sint()`* and *`read_uint()`* processes, except that instances of *`read_bool()`* shall be replaced by instances of *`read_boola()`*, defined in annex B.2.4, using a context for each bit.

*`read_sinta()`*and *`read_uinta()`* shall be provided with a map `context_prob_set` which shall consist of three parts as follows:

1. an array of follow context labels, `context_prob_set[FOLLOW]`,
2. a single data context index, `context_prob_set[DATA]` and
3. a sign context index, `context_prob_set[SIGN]` (ignored for unsigned integer decoding).

Each follow context probability shall be used for decoding the corresponding follow bit, with the last follow context being used for all subsequent follow bits also (if any).

The follow context selection function, *`follow_context()`* shall be defined as follows:

```
follow_context(index, context_prob_set):
  pos = min(index, length(context_prob_set[FOLLOW]) - 1)
  ctx_label = context_prob_set[FOLLOW][pos]
  return state[context_probs][ctx_label]
```

So the last follow context shall also be used for all the remaining follow bits.

A.4.3.2 Unsigned Integer Decoding

The *`read_uinta()`* function shall be defined as follows:

```
read_uinta(context_prob_set):
   value = 1
   index = 0
   while(read_boola(follow_context(index, context_prob_set)) == False):
      value <<= 1
      if(read_boola(state[context_probs][context_prob_set[data] ])):
         value += 1
      index += 1
   value -= 1
   return value
```

A.4.3.3 Signed Integer Decoding

Signed integers shall be defined as follows using the *read_sinta()* process. This process shall decode the magnitude first, then the sign, as necessary.

The *read_sinta()* function shall be as defined as follows:

```
read_sinta(context_prob_set):
   value = read_uinta(context_prob_set)
   if(value != 0):
      if(read_boola(state[contexts][context_prob_set[sign] ]) == True):
         value = -value
   return value
```

Annex B Arithmetic Coding (Normative)

This annex has three parts:

1. a description of the principles of arithmetic coding,
2. a definition of the arithmetic decoding engine used in VC-2 and
3. a description of a compatible arithmetic encoder.

B.1 Arithmetic Coding Principles (Informative)

This section provides an introduction to the principles underlying arithmetic coding. It briefly describes binary arithmetic coding; that is the coding of binary symbols, which is used in VC-2.

Arithmetic coding is an extremely powerful form of entropy coding, which can closely approximate the Shannon information limit for given data. Arithmetic encoding consists of a state machine that is fed with a sequence of symbols together with an estimate of each symbol's probability. For each input symbol the arithmetic coding engine updates its state and output a number of coded bits. The number of output bits for each input symbol depends on the internal state and on the current probabilities the symbols that are coded, and can range from zero to many bits.

The variable number of coded bits output for each input symbol complicates the implementation but is essential to the optimal nature of arithmetic coding. Consider a binary symbol *b*, with probabilities $p(b = \text{False}) = p_0$ and $p(b = \text{True}) = 1-p_0$. The binary entropy of the symbol *b* is the expected number of bits required to encode it, and is equal to:

$$entropy(b) = p_0 \log_2(1/p_0) + (1-p_0)\log_2(1/(1-p_0))$$

If *entropy(b)* is plotted against p_0, it can be seen that if p_0 is not equal to exactly 0.5, then $e(p_0) < 1$. This means that an optimal binary entropy encoder that operates symbol by symbol will not produce an output for every symbol.

B.1.1 Interval Division and Scaling

The fundamental idea of arithmetic coding is interval division and scaling. An arithmetic code can be thought of as a single number lying in an interval determined by the sequence of values having been coded.

Let us begin with the interval [0*,* 1), and let us suppose that we know (or have some estimate of) the probability of **`False`**, p_0. Conceptually we divide the interval into the sub-intervals [0*,* p_0) and [p_0, 1).

Suppose the first symbol to be coded is **`False`**, then the interval becomes [0*,* p_0). Alternatively if the first symbol to be coded is **`True`**, then the interval becomes [p_0, 1). After coding one or more symbols we arrive at an interval [*low, high*). To code the next symbol we partition this interval into [*low, low* + p_0(*high* - *low*)) and [*low* + p_0(*high* - *low*)*, high*]. If the symbol is **`False`** we choose the first interval, and if it is **`True`** the second interval.

For any integer *N*, this process clearly partitions the interval [0*,* 1) into a set of disjoint intervals that correspond to all the possible sequences of *N* bits. Identifying such a bit sequence is equivalent to choosing a value in the corresponding interval, and for an interval width *w* that in general requires

$$Ceiling(\log_2 1/w))$$

bits. With static probabilities, on average,

$$w = p_0^{Np_0}(1-p_0)^{N(1-p_0)}$$

resulting in

$$N.e(p_0)$$

bits being used, demonstrating the near optimal nature of arithmetic coding. Moreover, it is clearly possible to create an adaptive arithmetic code by changing the estimate of p_0 based on previously coded data.

B.1.2 Finite Precision Arithmetic

As it stands, the procedure outlined in the previous section has a number of drawbacks for practical application. Firstly, it requires unlimited precision to scale the interval, which is not available in real hardware or software. Secondly, it only produces an output when all values have been coded. These problems are addressed by re-normalization and progressive output: periodically rescaling the interval, and outputting the most significant bits of *low* and *high* whenever they agree.

For example, if we know that

low = *b*0*xyz*…

and

high = *b*0*pqr*…

then we can output zero, since this must prefix any value lying in the interval, and shift *low* and *high* to get

low = *bxyz*…

and

high = *bpqr*… .

This has the effect of doubling the interval from 0 ($x \rightarrow 2x$).

Likewise, if *low* = *b*1*xyz*… and *high* = *b*1*pqr*…, we can output 1 and shift to get *low* = *bxyz*… and *high* = *bpqr*… again: this is equivalent to doubling the interval from 1 ($x \rightarrow 2x$ - 1).

One problem remains: suppose the interval resolutely sits on the fence, straddling ½ whilst getting smaller and smaller, with the most significant bits of low and high staying as 0 and 1 respectively. In this case, when the straddle is finally resolved, *low* and *high* will both be of the form *b*10000…*xyz* or *b*01111…*pqr*.

The resolution strategy is to again rescale *low* and *high*, but this time double from ½ (i.e. $x \rightarrow 2x - ½$), and keep a count of the number *k* of times this is done, as this is the number of carry bits that are required. When the straddle is resolved as 1, then 1 followed by *k* zero bits is output, otherwise 0 followed by k 1's is output. This ensures that the output exactly represents the small straddling interval.

A decoder can determine a symbol as soon as it has sufficient bits to distinguish whether a value lies in one interval or another. If constraints are placed on the size of the smallest interval before renormalization (for example, by renormalizing often enough and by having a fixed smallest allowable probability), then this can be accomplished within a fixed word width.

B.1.3 Symbol Probability Estimation

The effectiveness of arithmetic coding is critically dependent on the accurate estimation of symbol probabilities. This might be achieved by counting the number of **`False`** symbols, n_0, and **`True`** symbols, n_1, that have previously been coded (or decoded), and estimating the probability of zero as:

$$p_0 = n_0/(n_0+n_1)$$

This requires maintaining the values of the two counts, n_0 and n_1, and performing some arithmetic (including division) every time a symbol is coded or decoded, in order to estimate the probability. Usually the values of n_0 and n_1 are both initialized to one for the first symbol, giving an initial probability estimate of 0.5.

With this method of probability estimation if **`False`** is coded the probability becomes

$p_0=n_0+1/(n_0+n_1+1)$,

and if **True** is coded the probability becomes

$p_0=n_0/(n_0+n_1+1)$.

In other words:

- If **False** is coded the probability update is: p_0 += $(1-p_0)/(n_0+n_1+1)$
- else, if **True** is coded, the probability update is: p_0 -= $p_0/(n_0+n_1+1)$

In VC-2 the probability estimate is simplified by assuming an estimate for the total number of symbols coded, t, as a function of probability itself. That is:

$$n_0+n_1 \bullet g(p_0)$$

With this simplifying assumption the probability update process becomes:

- If **False** is coded: p_0 += $(1-p_0)/(g(p_0)+1)$
- else, if **True** is coded: p_0 -= $p_0/(g(p_0)+1)$

Many choices would be possible for the definition of $g(p_0)$. For VC-2, $g(p_0)$ is given by:

$$g(p_0) \bullet 2^v \quad \text{where } v=8-4.\log_2(128.(1-2.\text{abs}(p_0-0.5)))/7$$

With this choice of $g(p_0)$, (n_0+n_1) is assumed to be 16 when p_0=0.5 and 256 when p_0=1/256 or 255/256.

Assuming the total number of symbols coded is a function of probability significantly simplifies the implementation. The probability update can be achieved with a single arithmetic operation (addition or subtraction) and a pair of lookup tables. That is:

- If **False** is coded: p_0 += LUT_False[p_0]
- else, if **True** is coded: p_0 -= LUT_True[p_0]

In the VC-2 arithmetic codec, probability is represented as a 16 bit unsigned integer (that is the range [0,65536) represents the range of probabilities [0,1)). To reduce the size of the lookup table only the 8 most significant bits of p_0 are used as the index to the lookup table. The definition of LUT_True is:

```
LUT_True(prob):
    lut_input &= 0xFF00
    if (lut_input >= 0x8000) p0 = lut_input/65536.0
    else p0 = (lut_input+256)/65536.0
    v = 8.0-4.0*log2(128.0*(1.0-2.0*abs(p0-0.5)))/7.0
    t = 2**v
    return round(lut_input+128)/t+1)
```

Here prob is the 16bit representation of probability used in the VC-2 arithmetic codec; the return value is in the same format. The 8 LSBs of prob are ignored allowing the use of a 256 entry lookup table. p0 is calculated using a dead-zone quantiser. The lookup table for False symbols is calculated similarly. It turns out that LUT_False is the mirror image of LUT_True so the same table can be used for both.

Note: This pseudocode does not conform to the definitions in section 4 because it contains floating point numbers,

floating point division and the round function.

B.2 Arithmetic Decoding Engine (Normative)

This section is a normative specification of the operation of the arithmetic decoding engine and the processes for using it to extract binary values from coded streams.

The arithmetic decoding engine shall consist of two elements:

1. a collection of state variables representing the state of the arithmetic decoder.
2. a function for extracting binary values from the decoder and updating the decoder state.

B.2.1 State and Contexts

The arithmetic decoder state shall consist of the following decoder state variables:

- **state**[low], an integer representing the beginning of the current coding interval.
- **state**[range], an integer representing the size of the current coding interval.
- **state**[code], an integer within the interval from **state**[low] to **state**[low]+ **state**[range], determined from the encoded bitstream.
- **state**[bits_left], a decrementing count of the number of bits yet to be read in.
- **state**[context_probs], a map of all the context probabilities used in the VC-2 decoder.

A context probability prob shall be a 16 bit unsigned integer value which encapsulates the probability of a zero symbol in the stream, such that

$$0 < prob \leq 0xFFFF$$

Contexts shall be accessed by decoding functions via a context label passed to the decoding functions.

B.2.2 Initialization

At the beginning of the decoding of any data unit, the arithmetic decoding state shall be initialized as follows:

```
initialize_arithmetic_decoding(ctx_labels):
    state[low] = 0x0
    state[range] = 0xFFFF
    state[code] = 0x0
    for i = 0 to 15:
        state[code] <<= 1
        state[code] += read_bitb()
    init_context_probs(ctx_labels)
```

The *init_context_probs*() function shall be defined as follows:

```
init_context_probs(ctx_labels):
    for i=0 to length(ctx_labels)-1:
        state[contexts][ctx_labels[i]] = 0x8000
```

Note: The value 0x8000 represents probability ½, or equal likelihood for binary values.

B.2.3 Data Input

The arithmetic decoding process shall access data in a contiguous block of bytes whose size shall be set on initialization. The bits in this block are sufficient to allow for the decoding of all coefficients. However, the specification of arithmetic decoding operations in this section may occasionally cause further bits to be read, even though they are not required for determining decoded values. For this reason the bounded-block read function *read_bitb()* (Annex A.3.1) shall be used for data access.

Since the length of arithmetically coded data elements is given in bytes within the VC-2 stream, there may be bits left unread when all values have been extracted. These shall be flushed by the *flush_inputb()* process defined in Annex A.3.1. This byte-aligns data after each arithmetically coded block.

Note: The VC-2 arithmetic decoding engine uses 16 bit words, and so if all coefficients are coded no more than 16 additional bits need be read beyond the end of the block.

B.2.4 Decoding Boolean Values

The arithmetic decoding engine is a multi-context, adaptive binary arithmetic decoder, performing binary re-normalization and producing binary outputs. For each bit decoded, the semantics of the relevant calling decoder function shall determine which contexts are passed to the arithmetic decoding operations.

This section defines the operation of the *read_boola()* function for extracting a boolean value from the VC-2 stream.

The decoding process for extracting a boolean value shall be as follows:

***read_boola*(context_label):**	**Ref**
prob_zero = **state**[contexts_probs][context_label]	
count = **state**[code] - **state**[low]	
range_times_prob = (**state**[range] * prob_zero) >> 16	
if(count >= range_times_prob):	
value = **True**	
state[low] += range_times_prob	
state[range] -= range_times_prob	
else:	
value = **False**	
state[range] = range_times_prob	
update_context_prob(**state**[contexts][context_index], value)	B.2.6
while (**state**[range] <= 0x4000):	
renormalise()	B.2.5
return *value*	

Note: The function above scales the probability of the symbol 0 from the decoding context so that if this probability were 1, then the interval would equal that between:

state[low] and **state**[low] + **state**[range] - 1

and count is set to the normalized cut-off between 0 and 1 within this new range.

B.2.5 Renormalization

Renormalization shall be applied to prevent the arithmetic decoding engine from losing accuracy.

Renormalization shall be applied while the range is less than or equal to a quarter of the total available 16-bit range (0x4000).

Renormalization shall double the interval and read a bit into the codeword.

The *renormalize()* function shall be defined as follows:

```
renormalize():
   if(((state[low] + state[range] - 1) ^ state[low]) >= 0x8000):
      state[code] ^= 0x4000
      state[low] ^= 0x4000
   state[range] <<= 1
   state[low] <<= 1
   state[low] &= 0xFFFF
   state[code] <<= 1
   state[code] += read_bitb()
   state[code] &= 0xFFFF
```

Note: For convenience let:

low = **state**[low] and *high* = **state**[low] + **state**[range] - 1

represent the upper and lower bounds of the interval. If the range is **<=** 0x4000, then one of the three following possibilities will be obtained:

1. the MSBs of *low* and *high* are both 0
2. the MSBs of *low* and *high* are both 1
3. *low* = *b*01..., *high* = *b*10..., and the interval straddles the half-way point (0x8000).

The re-normalization process has the effect that:

- in case 1, the interval [*low, high*] is doubled from 0 ($x \rightarrow 2 * x$),
- in case 2, it is doubled from 1 ($x \rightarrow 2 * x - 1$); and
- in case 3, it is doubled from $^1/_2$ ($x \rightarrow 2x - 0.5$).

B.2.6 Updating Context Probabilities

Context probabilities shall be updated according to a probability look-up table: **state**[lut] which supplies a value for decrementing or incrementing the probability of zero based on the first 8 bits of its current value, according to Table B.1 (see the end of this section). The *update_context_prob()* process shall be defined as follows:

update_context_prob(ctx_prob, value):	
if(value == **True**):	
ctx_prob -= **state**[lut][ctx_prob >> 8]	Table B.1
else:	
ctx_prob += **state**[lut][255 - (ctx_prob >> 8)]	Table B.1

The lookup table used for updating context probabilities shall be as defined in Table B.1. The lookup table entries are arranged in raster scan order with rows of thirteen entries. The entry corresponding to index zero is in the top left hand corner, the index increments by one from left to right and by thirteen from top to bottom, the entry corresponding to index 255 is on the right hand side of the last row.

Table B.1 – Look-up table for context probability adaptation

state[lut][] (indexes 0 to 255)
0, 2, 5, 8, 11, 15, 20, 24, 29, 35, 41, 47, 53,
60, 67, 74, 82, 89, 97, 106, 114, 123, 132, 141, 150, 160,
170, 180, 190, 201, 211, 222, 233, 244, 256, 267, 279, 291, 303,
315, 327, 340, 353, 366, 379, 392, 405, 419, 433, 447, 461, 475,
489, 504, 518, 533, 548, 563, 578, 593, 609, 624, 640, 656, 672,
688, 705, 721, 738, 754, 771, 788, 805, 822, 840, 857, 875, 892,
910, 928, 946, 964, 983, 1001, 1020, 1038, 1057, 1076, 1095, 1114, 1133,
1153, 1172, 1192, 1211, 1231, 1251, 1271, 1291, 1311, 1332, 1352, 1373, 1393,
1414, 1435, 1456, 1477, 1498, 1520, 1541, 1562, 1584, 1606, 1628, 1649, 1671,
1694, 1716, 1738, 1760, 1783, 1806, 1828, 1851, 1874, 1897, 1920, 1935, 1942,
1949, 1955, 1961, 1968, 1974, 1980, 1985, 1991, 1996, 2001, 2006, 2011, 2016,
2021, 2025, 2029, 2033, 2037, 2040, 2044, 2047, 2050, 2053, 2056, 2058, 2061,
2063, 2065, 2066, 2068, 2069, 2070, 2071, 2072, 2072, 2072, 2072, 2072, 2072,
2071, 2070, 2069, 2068, 2066, 2065, 2063, 2060, 2058, 2055, 2052, 2049, 2045,
2042, 2038, 2033, 2029, 2024, 2019, 2013, 2008, 2002, 1996, 1989, 1982, 1975,
1968, 1960, 1952, 1943, 1934, 1925, 1916, 1906, 1896, 1885, 1874, 1863, 1851,
1839, 1827, 1814, 1800, 1786, 1772, 1757, 1742, 1727, 1710, 1694, 1676, 1659,
1640, 1622, 1602, 1582, 1561, 1540, 1518, 1495, 1471, 1447, 1422, 1396, 1369,
1341, 1312, 1282, 1251, 1219, 1186, 1151, 1114, 1077, 1037, 995, 952, 906,
857, 805, 750, 690, 625, 553, 471, 376, 255

Note: A single 512-element LUT taking `value` as an argument can avoid the branch in *update_context_prob*().

B.3 Arithmetic Encoding (Informative)

This document only normatively defines the decoding of arithmetic coded data. However whilst it is clearly vital that an encoding process matches the decoding process, it is not entirely straightforward to derive an implementation of the encoder by only considering the decoder specification. Therefore this informative section describes a possible implementation for an arithmetic encoder that will produce an output that is decodable by the VC-2 arithmetic decoder. This section is best read in conjunction with Annex B.2.

B.3.1 Encoder Variables

An arithmetic encoder requires the following unsigned integer variables:

- `low`, a value indicating the bottom of the encoding interval
- `range`, a value indicating the width of the encoding interval
- `carry`, a value tracking the number of unresolved straddle conditions (described below)
- a set of 16-bit context probabilities, as described in annex A.4.

The process for updating context probabilities is described in annex B.2.6

A VC-2 binary arithmetic encoder implementation codes a set of data in three stages:

1. Initialization,
2. Processing of all values and
3. Flushing.

B.3.2 Initialization

Initialization of the arithmetic encoder is very simple; the internal variables are set as:

```
low    = 0x0
range  = 0xFFFF
carry  = 0
```

With 16-bit accuracy, 0*x*FFFF corresponds to an interval width value of (almost) 1. All context probabilities are initialized to probability 1⁄2 (0x8000).

B.3.3 Encoding Binary Values

The encoding process for a binary value must precisely mirror that for the decoding process (Annex B.2.4), in particular the interval variables `low` and `range` must be updated in the same way.

Coding a boolean value consists of three sub-stages (in order):

1. scaling the interval `[low, low + range)`
2. updating contexts
3. re-normalizing and outputting data

B.3.3.1 Scaling the Interval

The integer interval `[low, low + range)` represents the real interval

$$[l, h) = [low/2^{16}, (low + range)/2^{16}).$$

In a given context with label `label`, the probability of zero can be extracted as:

```
prob_zero = state[context_probs][label]
```

If 0 is to be encoded, the real interval `[l, h)` is rescaled so that `l` is unchanged and the width $r = h-l = range/2^{16}$ is scaled to `r * p0` where $p0 = prob\_zero/2^{16}$. This operation is approximated by setting:

```
range = (range * prob_zero) >> 16
```

If 1 is to be encoded, `[l, h)` is rescaled so that h is unchanged and `r` is scaled to `(1 - p0) * r`.

This operation is approximated by setting:

```
low += (range * prob_zero) >> 16
range -= (range * prob_zero) >> 16
```

B.3.3.2 Updating Contexts

Contexts are updated in exactly the same way as the decoder (Annex B.2.6).

B.3.3.3 Renormalization and Output

Renormalization must cause `low` and `range` to be modified exactly as in the decoder (Annex B.2.5). In addition, during renormalization, bits are output when `low` and `low + range` agree in their MSBs, taking into account carries accumulated when a straddle condition is accumulated.

In pseudo code, this is as follows:

```
…
  while(range <= 0x4000):
     if (((low+range-1) ^ low) >= 0x8000):
        low ^= 0x4000
        carry += 1
     else:
        write bit( (low & 0x8000) != 0)
        while(carry > 0):
           write bit((low & 0x8000) == 0)
           carry -= 1
     low <<= 1
     range <<= 1
     low &= 0xFFFF
```

B.3.4 Flushing the Encoder

After encoding, there may still be insufficient bits for a decoder to determine the final few encoded symbols, partly because further renormalization is required (for example, MSBs may agree but the range may still be larger than 0x4000) and partly because there may be unresolved carries.

The following four steps will adequately flush the encoder:

1. output the remaining resolved MSBs,
2. resolve the remaining straddle conditions,
3. flush the carry bits,
4. byte align the output with padding bits.

The remaining MSBs are then output as follows:

```
…
```

```
   while(((low + range -1) ^ low) < 0x8000 ):
      write bit( (low & 0x8000) != 0)
      while(carry > 0):
         write bit((low & 0x8000) == 0x0)
         carry -= 1
      low <<= 1
      low &= 0xFFFF
      range <<= 1
```

Remaining straddles can then be resolved by:

```
…
   while((low & 0x4000) != 0 and
                          (((low + range - 1) & 0x4000) == 0x0)):
      carry += 1
      low ^= 0x4000
      low <<= 1
      range <<= 1
      low &= 0xFFFF
```

and carry bits can be discharged by picking a resolution of the final straddles:

```
…
   write bit((low & 0x4000) != 0)
   for c = 0 to carry:
      write bit((low & 0x4000) == 0)
```

Finally, 0-7 padding bits are added to the encoded output to make a whole number of bytes. These are not necessary for decoding, but for stream compliance.

Annex C Predefined Video Formats (Normative)

This annex defines the default values of video parameters that are determined by the value of the base video format. These defaults reduce overhead by allowing a large number of parameters to be set without explicit signaling.

The collection of default values for each value of the base video format constitutes a map, which shall be returned by the *set_source_defaults*(base_video_format) function and used as a basis for defining the source video format in the sequence header as per Section 11.3.

All source parameters for any of the predefined video formats may be overridden as required in the sequence header. In the specific case of the custom Video Format 0, all the values shall be considered as default values to be over-ridden as needed to specify the custom format.

Table C.1a – Default source parameters for video formats 0~8

Video Formats									
Base Video Format index value:	0	1	2	3	4	5	6	7	8
Name (Informative):	Custom Format	QSIF525	QCIF	SIF525	CIF	4SIF525	4CIF	SD480I-60	SD576I-50
Frame width:	640	176	176	352	352	704	704	720	720
Frame height:	480	120	144	240	288	480	576	480	576
Color difference sampling format:	4:2:0	4:2:0	4:2:0	4:2:0	4:2:0	4:2:0	4:2:0	4:2:2	4:2:2
Source sampling:	0	0	0	0	0	0	0	1	1
Top field first:	False	False	True	False	True	False	True	False	True
Frame rate index:	1	9	10	9	10	9	10	4	3
numerator:	24000	15000	25	15000	25	15000	25	30000	25
denominator:	1001	1001	2	1001	2	1001	2	1001	1
Pixel aspect ratio index:	1	2	3	2	3	2	3	2	3
numerator:	1	10	12	10	12	10	12	10	10
denominator:	1	11	11	11	11	11	11	11	11
Clean width:	640	176	176	352	352	704	704	704	704
Clean height:	480	120	144	240	288	480	576	480	576
Clean left offset:	0	0	0	0	0	0	0	8	8
Clean top offset:	0	0	0	0	0	0	0	0	0
Signal range index:	1	1	1	1	1	1	1	3	3
Luma offset:	0	0	0	0	0	0	0	64	64
Luma excursion:	255	255	255	255	255	255	255	876	876

Color difference offset:	128	128	128	128	128	128	128	512	512
Color difference excursion:	255	255	255	255	255	255	255	896	896
Color specification index:	0 Custom	1 SDTV 525	2 SDTV 625	1 SDTV 525	2 SDTV 625	1 SDTV 525	2 SDTV 625	1 SDTV 525	2 SDTV 625
Color primaries index:	0	1	2	1	2	1	2	1	2
Description (informative)	HDTV	SDTV 525	SDTV 625	SDTV 525	SDTV 625	SDTV 525	SDTV 625	SDTV 525	SDTV 625
Color matrix index:	0	1	1	1	1	1	1	1	1
Description (informative):	HDTV	SDTV	SDTV	SDTV	SDTV	SDTV	SDTV	SDTV	SDTV
Transfer function index:	0	0	0	0	0	0	0	0	0
Description (informative):	TV Gamma	TV Gamma	TV Gamma	TV Gamma	TV Gamma	TV Gamma	TV Gamma	TV Gamma	TV Gamma

Table C.1b – Default source parameters for video formats 9~16

Video Formats								
Base Video Format index value:	9	10	11	12	13	14	15	16
Name (Informative):	HD720P-60	HD720P-50	HD1080I-60	HD1080I-50	HD1080P-60	HD1080P-50	DC2K	DC4K
Frame width:	1280	1280	1920	1920	1920	1920	2048	4096
Frame height:	720	720	1080	1080	1080	1080	1080	2160
Color difference sampling format:	4:2:2	4:2:2	4:2:2	4:2:2	4:2:2	4:2:2	4:4:4	4:4:4
Source sampling:	0	0	1	1	0	0	0	0
Top field first:	True	True	True	True	True	True	True	True
Frame rate index:	7	6	4	3	7	6	2	2
numerator:	60000	50	30000	25	60000	50	24	24
denominator:	1001	1	1001	1	1001	1	1	1
Pixel aspect ratio index:	1	1	1	1	1	1	1	1
numerator:	1	1	1	1	1	1	1	1
denominator:	1	1	1	1	1	1	1	1

Clean width:	1280	1280	1920	1920	1920	1920	2048	4096
Clean height:	720	720	1080	1080	1080	1080	1080	2160
Clean left offset:	0	0	0	0	0	0	0	0
Clean top offset:	0	0	0	0	0	0	0	0
Signal range index:	3	3	3	3	3	3	4	4
Luma offset:	64	64	64	64	64	64	256	256
Luma excursion:	876	876	876	876	876	876	3504	3504
Color difference offset:	512	512	512	512	512	512	2048	2048
Color difference excursion:	896	896	896	896	896	896	3584	3584
Color specification index:	3	3	3	3	3	3	4	4
Description (informative):	HDTV	HDTV	HDTV	HDTV	HDTV	HDTV	D-Cinema	D-Cinema
Color primaries index:	0	0	0	0	0	0	3	3
Description (informative):	HDTV	HDTV	HDTV	HDTV	HDTV	HDTV	D-Cinema	D-Cinema
Color matrix index:	0	0	0	0	0	0	3	3
Description (informative):	HDTV	HDTV	HDTV	HDTV	HDTV	HDTV	Reversible	Reversible
Transfer function index:	0	0	0	0	0	0	3	3
Description (informative):	TV Gamma	TV Gamma	TV Gamma	TV Gamma	TV Gamma	TV Gamma	D-Cinema Transfer Function	D-Cinema Transfer Function

Table C.1c – Default source parameters for video formats 17~22

Video Formats						
Base Video Format index value:	17	18	19	20	21	22
Name (Informative):	UHDTV 4K-60	UHDTV 4K-50	UHDTV 8K-60	UHDTV 8K-50	HD1080P-24	SD Pro486
Frame width:	3840	3840	7680	7680	1920	720
Frame height:	2160	2160	4320	4320	1080	486
Color difference sampling format:	4:2:2	4:2:2	4:2:2	4:2:2	4:2:2	4:2:2
Source sampling:	0	0	0	0	0	1
Top field first:	True	True	True	True	True	False

Frame rate index:	7	6	7	6	1	4
numerator:	60000	50	60000	50	24000	30000
denominator:	1001	1	1001	1	1001	1001
Pixel aspect ratio index:	1	1	1	1	1	2
numerator:	1	1	1	1	1	10
denominator:	1	1	1	1	1	11
Clean width:	3840	3840	7680	7680	1920	720
Clean height:	2160	2160	4320	4320	1080	486
Clean left offset:	0	0	0	0	0	0
Clean top offset:	0	0	0	0	0	0
Signal range index:	3	3	3	3	3	3
Luma offset:	64	64	64	64	64	64
Luma excursion:	876	876	876	876	876	876
Color difference offset:	512	512	512	512	512	512
Color difference excursion:	896	896	896	896	896	896
Color specification index:	3	3	3	3	3	3
Description (informative):	HDTV	HDTV	HDTV	HDTV	HDTV	HDTV
Color primaries index:	0	0	0	0	0	0
Description (informative):	HDTV	HDTV	HDTV	HDTV	HDTV	HDTV
Color matrix index:	0	0	0	0	0	0
Description (informative):	HDTV	HDTV	HDTV	HDTV	HDTV	HDTV
Transfer function index:	0	0	0	0	0	0
Description (informative):	TV Gamma	TV Gamma	TV Gamma	TV Gamma	TV Gamma	TV Gamma

Annex D Profiles and Levels (Normative)

A VC-2 decoder shall support one or more different profiles and levels. Profiles and levels determine which tools, syntax elements and structures shall be supported, and what decoder resources (computational and memory) are required.

D.1 Profiles

A given profile requires that particular syntax/syntax elements shall be used and that decoder variables or functions shall be set to particular values.

VC-2 defines three profiles, main, simple and low delay, corresponding to different picture types. These shall satisfy the following conditions:

- A low delay profile VC-2 sequence shall set **`state`**`[profile]` equal to a value of 0 in the parse parameters (Section 10.4) for each sequence header in the sequence.
- A simple profile sequence shall set **`state`**`[profile]` equal to a value of 1 in the parse parameters for each sequence header in the sequence.
- A main profile sequence shall set **`state`**`[profile]` equal to a value of 2 in the parse parameters for each sequence header in the sequence.

A VC-2 sequence shall comply with one of the supported profiles.

D.1.1 Low Delay Profile

A low delay profile sequence shall contain only those data units whose parse codes are listed in Table D.1.

Table D.1 – Parse code values for low delay profile sequences

Parse Code (hex)	Binary code (informative)	Description
0x00	0000 0000	Sequence header
0x10	0001 0000	End of sequence
0x20	0010 0000	Auxiliary data
0x30	0011 0000	Padding data
Low delay syntax:		
0xC8	1100 1000	Picture (no arithmetic coding)

A low delay profile sequence shall contain no pictures using the core syntax.

D.1.2 Simple Profile

A simple profile sequence shall contain only those data units whose parse codes are listed in Table D.2.

Table D.2 – Parse code values for simple profile sequences

Parse Code (hex)	Binary code (informative)	Description
0x00	0000 0000	Sequence header
0x10	0001 0000	End of sequence
0x20	0010 0000	Auxiliary data
0x30	0011 0000	Padding data
Core syntax:		
0x48	0100 1000	Picture (no arithmetic coding)

A simple profile sequence shall contain core syntax pictures that do not use arithmetic coding.

D.1.3 Main Profile

A main profile sequence may contain any of the data units whose parse codes are listed in Table D.3.

Table D.3 – Parse code values for main profile sequences

Parse Code (hex)	Binary code (informative)	Description
0x00	0000 0000	Sequence header
0x10	0001 0000	End of sequence
0x20	0010 0000	Auxiliary data
0x30	0011 0000	Padding data
Core syntax:		
0x08	0000 1000	Picture (arithmetic coding)

A main profile sequence shall contain core syntax pictures that use arithmetic coding.

D.2 Levels

A given value of level shall define constraints on the decoder resources required to decode a compliant sequence. The parameter values (or range of values) that are constrained for each level shall be defined in associated SMPTE documents. Level definitions may include specific limits on the coded bit rates defined typically as limits on the number of bits per picture or bits per slice.

Note: Generalized level values are defined in the companion standard SMPTE 2042-2. Specialized level values are defined in other associated SMPTE documents.

Annex E Low Delay Quantization Matrices (Normative)

This annex defines the default quantization tables to be used in the low delay syntax and provides an informative description of quantization matrix design principles.

E.1 Default Quantization Matrices

This section defines default quantization matrices to be used for the quantization of slice coefficients in the low delay syntax.

The following tables define matrices for values of **state**[dwt_depth] between 0 and 4 inclusive.

Values of **state**[dwt_depth] not present in the tables in this section shall require a custom matrix to be encoded, as per Section 12.3.4.2.

Informative advice for constructing custom quantization matrices based on noise power conservation and perceptual weighting is given in Annex E.2.

Table E.1 – Default quantization matrices for state[wavelet index] == 0 (Deslauriers-Dubuc (9,7))

		state[wavelet_depth]				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	LL	0	5	5	5	5
1	HL, LH , HH	-	3, 3, 0	3, 3, 0	3, 3, 0	3, 3, 0
2	HL, LH, HH	-	-	4, 4, 1	4, 4, 1	4, 4 ,1
3	HL, LH, HH	-	-	-	5, 5 ,2	5, 5, 2
4	HL, LH, HH	-	-	-	-	6, 6, 3

Table E.2 – Default quantization matrices for state[wavelet index] == 1 (LeGall (5,3))

		state[wavelet_depth]				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	LL	0	4	4	4	4
1	HL, LH , HH	-	2, 2, 0	2, 2, 0	2, 2, 0	2, 2, 0
2	HL, LH, HH	-	-	4, 4, 2	4, 4, 2	4, 4, 2
3	HL, LH, HH	-	-	-	5, 5, 3	5, 5, 3
4	HL, LH, HH	-	-	-	-	7, 7, 5

Table E.3 – Default quantization matrices for state[wavelet index] == 2 (Deslauriers-Dubuc (13,7))

		state[wavelet_depth]				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	LL	0	5	5	5	5
1	HL, LH , HH	-	3, 3, 0	3, 3, 0	3, 3, 0	3, 3, 0
2	HL, LH, HH	-	-	4, 4, 1	4, 4, 1	4, 4, 1
3	HL, LH, HH	-	-	-	5, 5, 2	5, 5, 2
4	HL, LH, HH	-	-	-	-	6, 6, 3

Table E.4 – Default quantization matrices for `state[wavelet index]` == 3 (Haar with no shift)

		state[wavelet_depth]				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	LL	0	8	12	16	20
1	HL, LH , HH	-	4, 4, 0	8, 8, 4	12, 12 ,8	16, 16, 12
2	HL, LH, HH	-	-	4, 4, 0	8, 8, 4	12, 12, 8
3	HL, LH, HH	-	-	-	4, 4, 0	8, 8, 4
4	HL, LH, HH	-	-	-	-	4, 4, 0

Table E.5 – Default quantization matrices for `state[wavelet index]` == 4 (Haar with single shift per level)

		state[wavelet_depth]				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	LL	0	8	8	8	8
1	HL, LH , HH	-	4, 4, 0	4, 4, 0	4, 4, 0	4, 4, 0
2	HL, LH, HH	-	-	4, 4, 0	4, 4, 0	4, 4, 0
3	HL, LH, HH	-	-	-	4, 4, 0	4, 4, 0
4	HL, LH, HH	-	-	-	-	4, 4, 0

Table E.6 – Default quantization matrices for `state[wavelet index]` == 5 (Fidelity)

		state[wavelet_depth]				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	LL	0	0	0	0	0
1	HL, LH , HH	-	4, 4, 8	4, 4, 8	4, 4, 8	4, 4, 8
2	HL, LH, HH	-	-	8, 8, 12	8, 8, 12	8, 8, 12
3	HL, LH, HH	-	-	-	13, 13, 17	13, 13, 17
4	HL, LH, HH	-	-	-	-	17, 17, 21

Table E.7 – Default quantization matrices for `state[wavelet index]` == 6 (Daubechies (9,7))

		state[wavelet_depth]				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	LL	0	3	3	3	3
1	HL, LH , HH	-	1, 1, 0	1, 1, 0	1, 1, 0	1, 1, 0
2	HL, LH, HH	-	-	4, 4, 2	4, 4, 2	4, 4, 2
3	HL, LH, HH	-	-	-	6, 6, 5	6, 6, 5
4	HL, LH, HH	-	-	-	-	9, 9, 7

E.2 Quantization Matrix Design and Quantizer Selection (Informative)

This section provides an informative guide to the principles used to design the default quantization matrix

E.2.1 Noise Power Normalization

The quantization matrices defined in the preceding section are designed to counteract the differential power gain of the various wavelet filters, so that quantization noise from each subband is weighted equally in terms of its contribution to noise power when transformed back into the picture domain.

Let α and β represent the RMS noise gain factors of the low-pass and high-pass wavelet filters used in

wavelet decomposition (see Annex G for a description of wavelet filtering). In the terminology of annex G where *C* and *D* are the low-pass and high-pass synthesis filters, the gain factors should be set so that:

$$\alpha = \left(\sum_n C(n)^2 \right)^{1/2} \text{ and } \beta = \left(\sum_n D(n)^2 \right)^{1/2}$$

(In this standard, wavelet synthesis filters have been defined in terms of lifting stages, which are filters operating on sub-sampled data. Wavelet filters are more conventionally represented in terms of an iterated binary filter bank: the relationship between these representations and how the lifting filters determine the filter bank is described in Annex G.)

In a single level of wavelet decomposition, quantization noise in each of the four subbands is therefore weighted by the factors shown in Figure E.1.

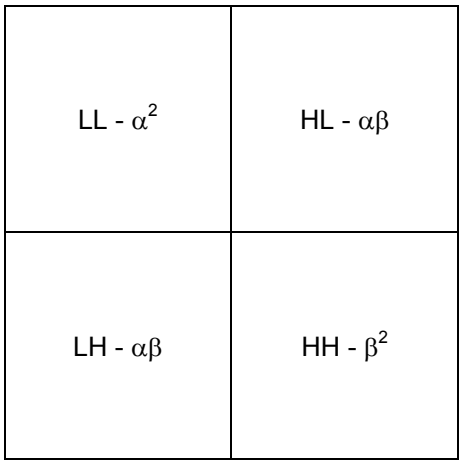

Figure E.1 – Subband weights for a 1-level decomposition

For higher levels of decomposition, these subband weighting factors iterate in the same manner as the wavelet transform itself. For example, with a 1-level decomposition, the first level LL band, with weight α^2 is further decomposed to give four more bands with weights as for the 1-level decomposition, but multiplied by α^2. This yields the weights shown in Figure E.2.

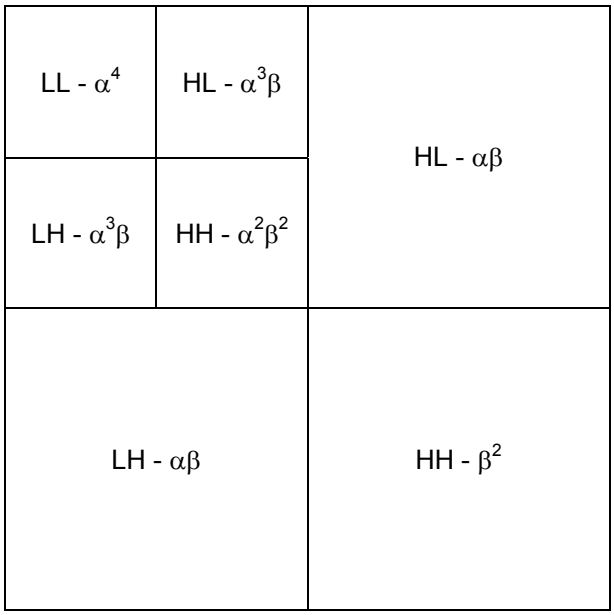

Figure E.2 – Subband weights for a 2-level decomposition

The quantization offset for a subband is determined from the normalized power gain for that subband. Normalization is performed by dividing by the smallest power gain of all subbands, and ensures that the smallest quantization offset is zero. The actual quantization offset is determined from the normalized subband power gain, w, by computing $4*\log_2(w)$ rounded to the nearest integer. The tables in the preceding section were produced using this process.

Note also that these factors must also take into account the shift factors used to add accuracy bits prior to each wavelet decomposition stage. For a filter shift of d, α and β are each multiplied by $2^{-d/2}$.

E.2.2 Custom Quantization Matrices

Custom matrices can also be defined that take into account not only noise power normalization but also other factors, for example perceptual weighting based on spatial frequency: additional multiplicative factors are computed for each subband, which produces a matrix of quantization offsets which can then be added to the default unweighted quantization matrices to produce a weighted quantization matrix.

An example perceptual weighting may be constructed from the ITU-R BT.959-2 Contrast Sensitivity Function (CSF). This is a function `csf(s)` which produces a value representing the sensitivity to detail at a given normalized spatial frequency *s*.

For luma, this is defined by:

$$csf(s) = 0.255 * (1 + 0.2561 * s^2)^{-0.75}$$

Assuming an isotropic response, we can form a 2-d perceptual weighting function on horizontal and vertical spatial frequencies s_x, s_y by:

$$CSF(s_x, s_y) = csf((s_x^2 + s_y^2)^{1/2}$$

Each subband in a wavelet decomposition represents a subset of spatial frequencies according to level and orientation, partitioning the spatial frequency domain as per Figure 7.3. Note that this partitioning is not normalized, since output pictures (and their compression artifacts) can be viewed over a range of distances.

Accordingly we can pick a representative, un-normalized horizontal and vertical spatial frequency ($f_x(b)$; $f_y(b)$, perhaps the middle frequency of the band. For example, an LH band *b* at level 1 in a 1-level decomposition will have mid frequency at (*pw*/4,3**ph*/4) where *ph* and *pw* are the padded width and height of the picture. This can be turned into a true spatial frequency by normalizing by the number of horizontal and vertical cycles per degree the output pictures will subtend at the target viewing distance and aspect ratio:

$$(f_x(b)/cpd_x, f_y(b)/cpd_y)$$

and this value fed into the weighting function *CSF* to get a value *c*(*b*). The appropriate quantization offset for that subband is then $4*\log_2(c(b))$, which can be used to modify the unweighted quantization matrix.

Annex F Video Systems Model (Informative)

F.1 Color Models

All current video systems use a YC_1C_2 form of coding of the RGB source values (Y,C_B,C_R is a commonly used variant of YC_1C_2). Although Y,C_B,C_R, is widely used, VC-2 can support other color systems such as Y,C_O,C_G as defined by ITU-T H.264 AVC annex E. For this reason the non-luma components are generalized to the terms C_1 and C_2.

The R, G and B components are tri-stimulus values (e.g., candelas/meter^2). Their relationship to CIE XYZ tri-stimulus values can be derived from the set of primaries and white point defined in the color primaries part of the color specification below using the method described in SMPTE RP 177-1993. In this document the RGB values are normalized to the range [0,1], so that RGB = 1,1,1 represents the peak white of the display device and RGB = 0,0,0 represents black.

The E_R, E_G, E_B values, are related to the linear RGB values by non-linear transfer functions. Normally, these values also fall in the range [0,1], but in the case of extended gamut systems (such as ITU-R BT.1361), negative values can occur. The non-linear transfer function is typically performed in the camera and is specified in the transfer characteristic part of the appropriate color specification. For aesthetic and psycho-visual reasons, the encoding transfer function is not always the inverse of the decoding transfer function. In fact the combined effect of the encoding and decoding transfer functions is such that the rendering intent or end-to-end gamma of the system can vary between about 1.1 and 1.6 depending on viewing conditions. The rationale for this is given in “Digital Video and HDTV” by Charles Poynton, (2003, Morgan Kaufmann Publishers, ISBN 1-55860-792-7).

The non-linear E_R, E_G, E_B values are subject to a matrix operation (known as ‘non-constant’ luma coding), which transforms them into luma (E_Y) and color difference (EC_1 and EC_2) values. E_Y is normally limited to the normalized range [0,1] and the color difference values to the normalized range [-0.5, 0.5]. In this standard, the color difference components are not to be confused with the chroma signals used by composite television systems where the color difference signals are significantly reduced in both resolution and signal amplitude. The color difference components used in this standard can be sub-sampled, either horizontally, vertically, or both horizontally and vertically.

Note: The E values can be viewed as something of a mathematical abstraction. For example, in digital display devices, R, G and B values are specified in terms of integer levels which are derived from the luma and color difference values by direct operations subsuming and approximating all the real-number operations described here. Generally, these approximations cause loss through quantization of intermediate values, and the restriction of values to particular ranges also restricts the color gamut.

F.1.1 YC_BC_R Coding

The E_Y, EC_1, EC_2 values are mapped to values for Y, C_B, C_R. Typically they are mapped to an 8-bit unsigned integer [0, 255]. The way this mapping occurs is defined by the signal range parameters. It is these integer values that are actually output from a decoder. In order to display the decoded video, the inverse to the above operations must be performed to convert this data to E_Y, EC_B, EC_R, then to E_R, E_G, E_B values and finally to R, G and B.

F.1.2 YC_OC_G Coding

In the case of Y, C_O, C_G coding, a lossless direct integer transform can be used, so that, together with 4:4:4 sampling and lossless compression, VC-2 can support efficient lossless RGB coding.

F.1.3 Signal Range

The offset and excursion values are used to convert the integer-valued decoded luma and color difference data Y, C_1, C_2 to normalized intermediate values E_Y, EC_1, and EC_2 such that:

- The E_Y value is the Y value scaled so that the range

  `[luma_offset, luma_offset+luma_excursion]`

  is mapped to the range [0,1],

- The EC_1 and EC_2 values are the C_1 and C_2 scaled so that the range

  `[color_diff_offset - color_diff_excursion/2, color_diff_offset + color_diff_excursion/2]`

  is mapped to [-0.5,0.5].

Note that in video systems, values overshooting or undershooting these values are normally allowed.

In the case of Y, C_O, C_G coding, E_Y, EC_G, and EC_O are not calculated. Instead, direct integer conversion to RGB is done (note: excursion values will be ignored in this integer conversion.)

F.1.4 Color Primaries

The color primaries allow device dependent linear RGB color co-ordinates to be mapped to device independent linear CIE XYZ space. The primaries specified are the CIE (1931) XYZ chromaticity co-ordinates of the primaries and the white point of the device.

The color primary specification therefore allows exact color reproduction of decoded RGB values on different displays with different display primaries.

F.1.5 Color Matrix

For conventional Y, C_B, C_R coding, luma and color difference values E_Y, EC_1, EC_2 are used to derive E_R, E_G, E_B values by applying a matrix.

In the case of Y, C_O, C_G coding, E_Y, EC_1, EC_1 are directly computed from the integer Y, C_O and C_G values by the following recipe, whereby integer RGB I_R, I_G, I_B values are decoded by:

Y -= `luma_offset`, C_G -= `color_diff_offset`, C_O -= `color_diff_offset`

temp = Y - (C_G >>1)

I_G = temp + C_G , I_B = temp - (C_O >>1) , I_R = IB + C_O

These are scaled down by dividing by (255 << accuracy_bits) and clipped to give the E_R, E_G and E_B values.

If the inverse transform has been correctly applied prior to coding and lossless coding employed, then clipping will be unnecessary.

Note that this matrix implies that the color difference range is twice as large as the RGB range (and the luma range), since the color difference components involve subtraction. Although logically knowing the signal range and scaling signals is prior to performing matrixing; the matrix parameters are coded first in the Display Parameters in order to allow the signal ranges to be correctly determined in this case.

F.2 Transfer Characteristics

F.2.1 TV Transfer Characteristic

Standard and high definition systems for both 50Hz and 60Hz based systems use an encoding gamma value of 0.45 with a linear portion at the low end of the scale to avoid the need for infinite gain at the receiver. The gamma values are defined in ITU-R BT.601-6 for standard definition and in ITU-R BT.709 for high definition.

F.2.2 Extended Color Gamut

ITU-R BT.1361 (Worldwide Unified Colorimetry of Future TV Systems) defines a color system with an extended color gamut. Refer to ITU-R BT.1361 for details.

ISO/IEC 61966-2 (Extended RGB Color Space) defines another color system with an extended color gamut. Refer to IEC 61966-2-2:2003 for details.

Note that use of the full range of Y, C_1, C_2 values can create negative R, G or B values in this case. The original color gamut equations were designed around the CRT (cathode ray tube) device. Some flat panel displays are capable of displaying a wider color gamut resulting in the desire to extend the color gamut to maximize the impact of these displays.

F.2.3 Linear

A linear transfer characteristic has f(x)=x; i.e. $E_X = X$.

F.3 Frame Rate

The frame rate value encodes the intended rate at which frames are to be displayed at the output of the decoder. If the scan format value defines that the video is interlaced, then fields are displayed at double the frame rate, in the order specified by the `top_field_first` flag. If `interlaced` is false, then the lines of each frame are displayed directly.

F.4 Aspect Ratios And Clean Area

F.4.1 Pixel Aspect Ratio

The pixel aspect ratio value of an image is the ratio of the intended spacing of horizontal samples (pixels) to the spacing of vertical samples (picture lines) on the display device. Pixel aspect ratios are fundamental properties of sampled images because they determine the displayed shape of objects in the whole image. Failure to use the correct value of pixel aspect ratio will result in distorted images where circles will be displayed as ellipses.

Most HDTV standards and computer image formats are defined to have pixel aspect ratios that are exactly 1:1.

For a number NH of pixels per unit length and NV pixels per unit height, this ratio is 1/NH : 1/NV or NV : NH. For a video standard of WxH pixels displayed at 4:3 picture aspect ratio, NH=W/4 and NV=H/3.

F.4.1.1 Using Non-Square Pixel Aspect Ratios

The defined pixel aspect ratios are designed to give image aspect ratios for standard definition television operating with a standard 4:3 picture aspect ratio.

For 525-line video, defining a 704 x 480 picture with a 4:3 aspect ratio results in a H:V pixel aspect ratio of 10:11 (i.e. 480/3 : 704/4).

For 625-line video defining a 704 x 576 picture with a 4:3 aspect ratio results in a H:V pixel aspect ratio of 12:11 (i.e. 576/3 : 704/4).

If the intended image aspect ratio is 16:9, then the H:V pixel aspect ratios change accordingly to 40:33 for 525-line video and 16:11 for 625-line video.

The values specified above are widely, but not unanimously, agreed to be the correct values. Differences of viewpoint arise from how much of the available horizontal picture size of 720 Y pixels is intended for display.

Implementers are advised to use one of the default pixel aspect ratios. VC-2 does allow non-standard pixel aspect ratios, although many display devices could ignore them and default to using different (and possibly unsuitable) values.

F.4.2 Clean Area

The clean area is intended to define an area within which picture information is subjectively uncontaminated by all edge distortions and possible unintended picture content such as microphones appearing at the top of the picture. It could be appropriate to display the clean area rather than the whole picture, which can contain edge distortions or unintended content.

The top-left corner of the clean area has coordinates (`left_offset`, `top_offset`) and dimensions `clean_width` and `clean_height`.

The clean area and the pixel aspect ratio can be used to determine the displayed image aspect ratio (which is the ratio of the width of the intended display area to the height of the intended display area). Regardless of the size of the clean area for display purposes, the pixel aspect ratio is maintained in order to avoid any geometric distortion.

Given two separate sequences, with identical image aspect ratios, if the top left corner and bottom left corners of their clean apertures are coincident when displayed, then all the images will be exactly coincident. This is regardless of the actual pixel dimensions of the images or their clean areas. This allows sequences to be combined together appropriately if they are appropriately scaled.

Annex G Wavelet Decimation and Reconstruction Processes (Informative)

G.1 Overview of Wavelet Processing

Figure G.1 illustrates a single stage of a generalized wavelet decimation followed by reconstruction. The aim is to get perfect (or near-perfect) reconstruction of the output so that it is identical to the original input.

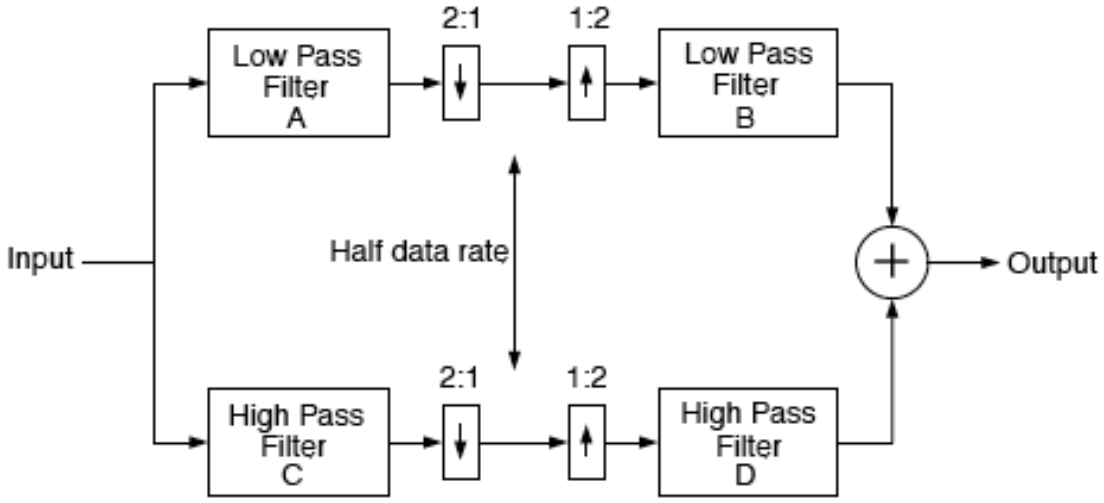

Figure G.1 –Single wavelet processing stage comprising decimation and reconstruction filters

The next figure (Figure G.2) illustrates how the frequency components are distributed both during decimation and reconstruction. This figures illustrates how the alias frequencies created during the decimation process are cancelled out during the reconstruction process. This feature of alias cancellation results from the wavelet process and is a specific attribute of wavelet coding. It is important to note that if the decoder receives imperfect signals (caused, for example, by quantization errors) then the imperfections will result in distortion in the reconstructed output.

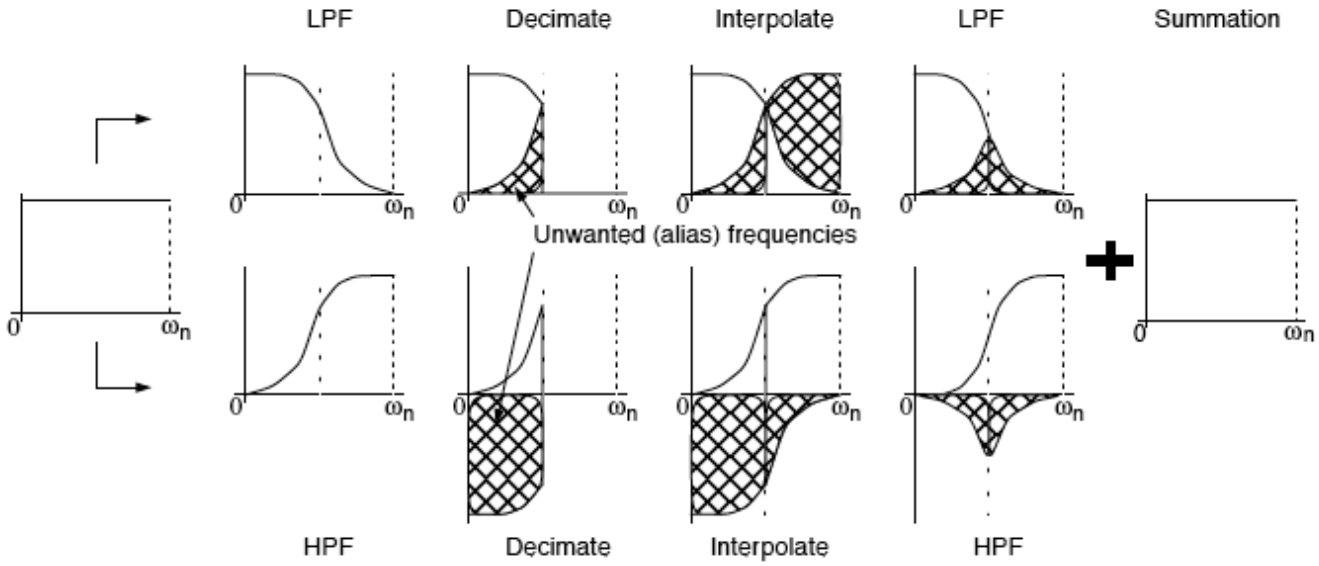

Figure G.2 – Illustration of the alias frequency generation and cancellation in a wavelet filter bank

A single wavelet stage is insufficient for most video coding applications. The figure below (G.3) illustrates how only the low-pass path is passed on to the next wavelet decimation step. Because each step of the wavelet decimation is self-contained, the reconstructed output is still identical to the input (barring quantization errors).

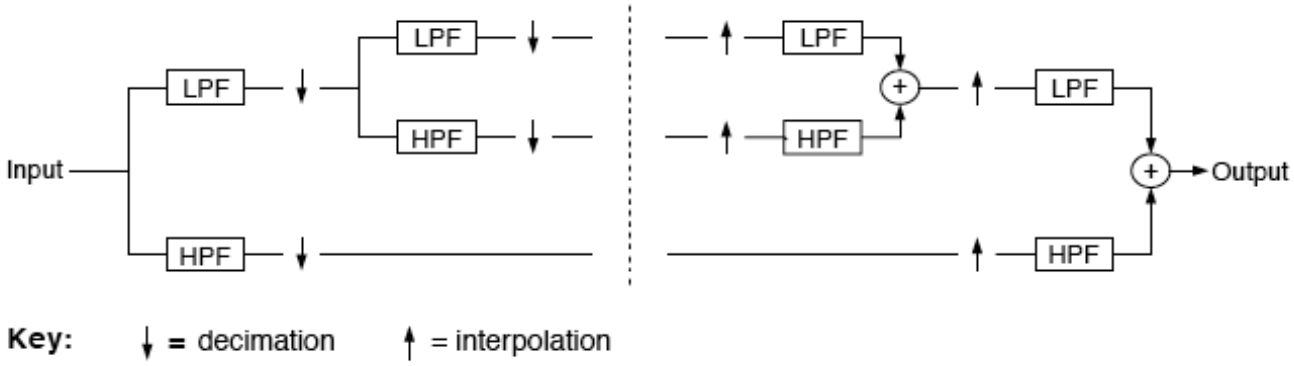

Figure G.3 –Two-step wavelet processing filter bank

The application of wavelet filter banks in picture coding results in a two-dimensional decimation process as illustrated in Figure G.4.

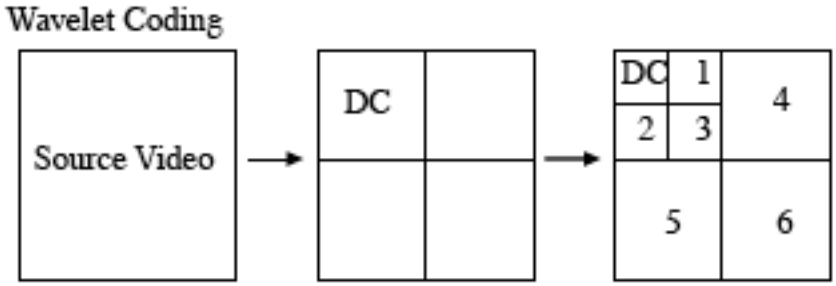

Figure G.4 –Decomposition of a single image into 7 wavelet frequency bands

Figure G.5 illustrates how a real image is decimated to produce a low-frequency proxy in the top-left corner and a range of increasing frequency band components extending to the right side for increasing horizontal frequencies and downwards for increasing vertical frequencies.

Figure G.5 –Decomposition of the EBU "Boats" picture into 7 wavelet frequency bands

G.2 The Lifting Process

This section provides a mathematical definition of how the lifting process can be applied to wavelet filter banks.

For any set of filters, the analysis and synthesis filter banks illustrated in Figure G.1 can be easily re-expressed as polyphase filter banks by means of applying *matrices* of filters in the sub-sampled domain. This is shown in Figure G.6, where $A(z)$ is the z–transform of the analysis polyphase filter matrix, and $S(z)$ is the z–transform of the synthesis polyphase filter matrix (the entries of both matrices being Laurent polynomials).

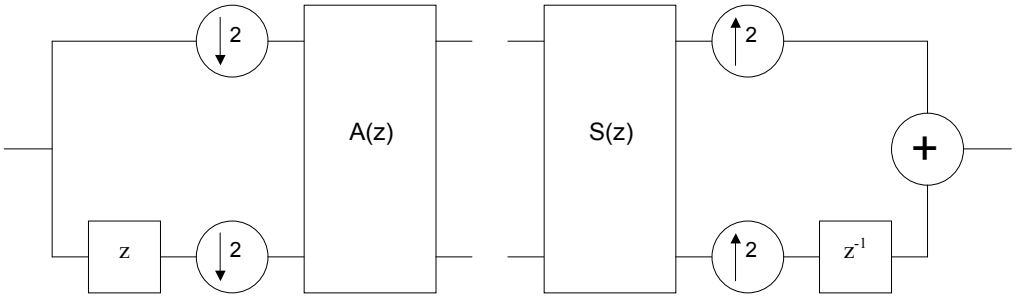

Figure G.6 – Polyphase representation of wavelet filter banks

In this representation, linear combinations of filters operate on both even and odd samples to produce new even and odd samples:

$$\begin{pmatrix} x_e^{out}(z) \\ x_o^{out}(z) \end{pmatrix} = A(z) \begin{pmatrix} x_e^{in}(z) \\ x_o^{in}(z) \end{pmatrix}$$

Since the filter process is invertible, it can be shown that the analysis and synthesis matrices are related by

$$\mathrm{A(z)} \;=\; (\mathrm{S}\,(\mathrm{z}^{-1})^{\mathrm{T}})^{-1}$$

Hence, in particular both the analysis and synthesis matrices are invertible. It can also be shown that this means that they are (up to gain factors and delays) factorizable into products of upper and lower triangular matrices as follows:

$$A(z) = \begin{pmatrix} 1 & a_1(z) \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ b_1(z) & 1 \end{pmatrix} \begin{pmatrix} 1 & a_2(z) \\ 0 & 1 \end{pmatrix} .....$$

Each upper or lower-triangular polyphase matrix represents a so-called *lifting* stage whereby either even coefficients are modified solely by odd coefficients or odd coefficients solely by even coefficients.

For example, if

$$\begin{pmatrix} x_e^{out}(z) \\ x_o^{out}(z) \end{pmatrix} = \begin{pmatrix} 1 & a(z) \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_e^{in}(z) \\ x_o^{in}(z) \end{pmatrix}$$

then

$$\begin{aligned} x_e^{out}(z) &= x_e^{in}(z) + a(z)x_o^{in}(z) \\ x_o^{out}(z) &= x_o^{in}(z) \end{aligned}$$

and the filter `a(z)` has been applied to the odd coefficients and then used to modify the even coefficients. Not only is this computationally efficient by breaking long filters into a number of shorter filters successively applied, but the factorization into such filter stages allows for computations to be done in-place.

Annex H Bibliography (Informative)

EBU Tech 3213-1994, Standard for Chromaticity Tolerances for Studio Monitors

ISO/IEC 15444-1:2002, JPEG 2000 Image Coding System

ITU-R Report BT.959-2 (1990), Experimental Results Relating Picture Quality to Objective Magnitude of Impairment

ISO/IEC 61966-2-2:2003, Multimedia Systems and Equipment — Colour Measurement and Management — Part 2-2: Colour Management — Extended RGB Colour Space — scRGB

ISO/IEC 646:1991, Information Technology - ISO 7-Bit Coded Character Set for Information Interchange

SMPTE ST 2042-2:2009, VC-2 Level Definitions

SMPTE RP 177-1993, Derivation of Basic Television Color Equations

“Digital Video and HDTV”, Charles Poynton 2003, Morgan Kaufmann Publishers, ISBN 1-55860-792-7

“Ripples in Mathematics, The Discrete Wavelet Transform”, A. Jenson and A.la Cour-Harbo, Springer, ISBN 3-540-41662-5

"Arithmetic coding revisited", Alasdair Moffat, ACM Transactions on Information Systems, Vol. 16 #3, July 1998